VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

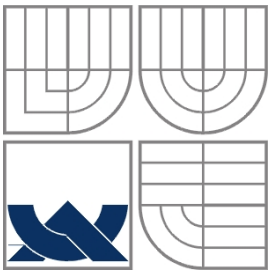
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# EXPLORATIVNÍ EDITACE ZDROJOVÝCH TEXTŮ

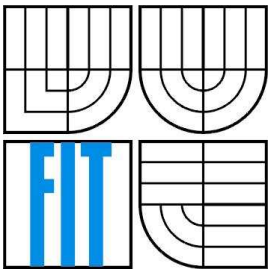
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE BC. IVAN ŠOMLO
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# EXPLORATIVNÍ EDITACE ZDROJOVÝCH TEXTŮ

EXPLORATIVE SOURCE CODE EDITOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. IVAN ŠOMLO

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ZBYNĚK KŘIVKA, PH.D.

BRNO 2011

*Zadání diplomové práce/12928/2010/xsomlo00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav informačních systémů Akademický rok 2010/2011

# Zadání diplomové práce

Řešitel: **Šomlo Ivan, Bc.**
Obor: Informační systémy
Téma: **Explorativní editace zdrojových textů**
**Explorative Source Code Editor**
Kategorie: Překladače

Pokyny:

1. Seznamte se se základními nástroji běžného explorativního editování zdrojového kódu v jazyce Smalltalk nebo SELF.
2. Navrhněte sadu podobných nástrojů (Class Browser, Hierarchy Browser, Inspector, ...) pro jiný objektově-orientovaný jazyk či sadu jazyků.
3. Dle pokynů vedoucího implementujte jádro potřebné pro podporu těchto nástrojů, které potom implementujte buď jako samostatné aplikace, nebo nejlépe integrujte do vhodného vývojového prostředí daného jazyka (např. Visual Studio, Eclipse, apod.).
4. Nástroje otestujte a popište budoucí možná rozšíření. Porovnejte výhody a nevýhody klasické editace a explorativní editace kódu.

Literatura:

- Křivánek, P., Křivka, Z.: Squeak Smalltalk, http://www.squeak.cz, (c)2003-2008.
- Sun Microsystems: Self Home Page, http://research.sun.com/self/, (c)1994-2009.
- Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Second Edition, 2006.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 20. září 2010
Datum odevzdání: 25. května 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
*vedoucí ústavu*

# Abstrakt

Diplomová práce se věnuje nástrojům, které jsou běžně používané vývojáři v explorativních prostředích jazyků Smalltalk a Self. Tyto jazyky osvobozují uživatele od nestrukturovaných zdrojových textů. Na druhé straně obsahují nástroje s bohatšími vlastnostmi. Práce obsahuje analýzu, návrh a popis implementace nástroje Object Viewer, který vychází z prostředí jazyka Self a je určený pro jazyky C a C++. Nástroj graficky zobrazuje fragmenty programu (objekty a ukazatele) a umožňuje jejich manipulaci během ladění programu. Na konci práce jsou shrnuty výsledky a navrhnuta další možná rozšíření.



# Abstract

The thesis discusses the tools commonly used by developers in exploratory environments of languages Smalltalk and Self. These languages free their users from using plain unstructured source code. Moreover, they contain tools with richer features. The thesis consists of an analysis, a design proposal, and a description of Object Viewer, a Self based tool designed for languages C and C++. The tool graphically displays fragments of a program (objects and pointers) and allows for manipulating them during debugging. The conclusion consists of a summary and a recommendation for additional new features.



# Klíčová slova

Explorativní programování, Smalltalk, Self, C++, vizualizace, diagram objektů, CodeModel, DIA, Visual Studio, Xrefs

# Keywords

Exploratory programming, Smalltalk, Self, C++, visualization, object diagram, CodeModel, DIA, Visual Studio, Xrefs


# Citace

Šomlo Ivan: Explorativní editace zdrojových textů, diplomová práce, Brno, FIT VUT v Brně, 2011

# Explorativní editace zdrojových textů

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ivan Šomlo
20.5.2011

## Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce Ing. Zbyňku Křivkovi, Ph.D. za cenné rady využitelné i mimo tuto práci a trvání na pravidelných hlášeních o stavu psaní práce.

# Obsah

# 1 Úvod

Konvenčný spôsob editovania zdrojových textov programov je založený na textovom editore a hierarchickej štruktúre zdrojových súborov. Tieto súbory obsahujú všetky definície tried a metód, komentáre a iné informácie. Súbory sa následne prekladajú kompilátorom. Výslednú podobu je možné spustiť a samozrejme ladiť. Ladiaci nástroj môže pri ladení zobrazovať obsah premenných a tiež ich na pokyn užívateľa meniť. Akákoľvek zmena v zdrojových textoch si vyžaduje ukončenie ladenia programu a jeho opätovné preloženie.

Existujú programovacie jazyky, ktoré zdrojové súbory nepoužívajú. Namiesto toho sa v nich rovno pracuje s logickými prvkami jazyka: s triedami, metódami, členskými premennými, objektmi a referenciami medzi nimi. V týchto jazykoch nie je jasne definované rozhranie medzi ladiacim a ladeným programom. Oba sú súčasťou toho istého prostredia. Taktiež nie je rozdiel medzi dobou kompilácie a dobou spustenia. Zmena v zdrojových textoch sa jednoducho aplikuje v dynamickom systéme okamžite. Tieto jazyky sa označujú ako exploratívne, pretože umožňujú experimentovať s vytváraným systémom za behu. Zvyčajne obsahujú sadu niekoľkých exploratívnych nástrojov, ktoré to umožňujú.

Ako to už býva, tieto dva extrémy na seba vplývajú. Výsledkom je, že exploratívne jazyky obsahujú funkcie na export a import zdrojových textov do/zo súborov. Na druhej strane moderné integrované vývojové prostredia pre konvenčné jazyky obsahujú presne tie isté exploratívne nástroje, ktoré sú bežné v exploratívnych jazykoch a dokonca umožňujú vykonávať niektoré zmeny v zdrojových textoch bez nutnosti reštartu ladeného systému. Táto práca obsahuje analýzu, návrh a popis implementácie jedného takého nástroja. Ide o nástroj Object Viewer, ktorý graficky zobrazuje živé objekty ladeného procesu.

Kapitola 2 obsahuje príklady existujúcich exploratívnych nástrojov. Ďalej obsahuje popis niektorých knižníc a rozhraní, ktoré sú použiteľné pri ich implementácií. Návrh samotného nástroja sa nachádza v kapitole 3. Kapitola 4 popisuje jednotlivé kroky pri implementácií. Dosiahnuté výsledky a návrh ďalšieho pokračovania projektu sú zhrnuté v kapitole 5. Kapitola 6 obsahuje záver.

# 2 Analýza

Táto kapitola obsahuje príklady niektorých nástrojov, ktoré sa bežne používajú pri programovaní v exploratívnych systémoch. Nasleduje analýza niekoľkých rozhraní, ktoré je možné použiť na vytvorenie podobných nástrojov aj pre konvenčné vývojové prostredia.

## 2.1 Exploratívne programovanie

Tradičný spôsob programovania vyžaduje, aby bolo od začiatku jasné aký systém sa implementuje. Takéto programovacie jazyky v sebe zahrňujú editáciu zdrojových súborov, ich kompiláciu a spustenie výsledného programu. Na druhej strane pri exploratívnom programovaní je možné vyvíjať systém postupne, bez znalosti celého systému. Takéto programovanie sa dosahuje v programovacích jazykoch väčšou interaktivitou, dynamickosťou a rozšíriteľnosťou jazyka a vývojových nástrojov [1]. Typickými zástupcami takýchto jazykov sú napríklad Lisp, Smalltalk a Self. Nasledujúce podkapitoly obsahujú príklady nástrojov, ktoré podporujú exploratívne programovanie v niektorých programovacích jazykoch alebo vývojových prostrediach.

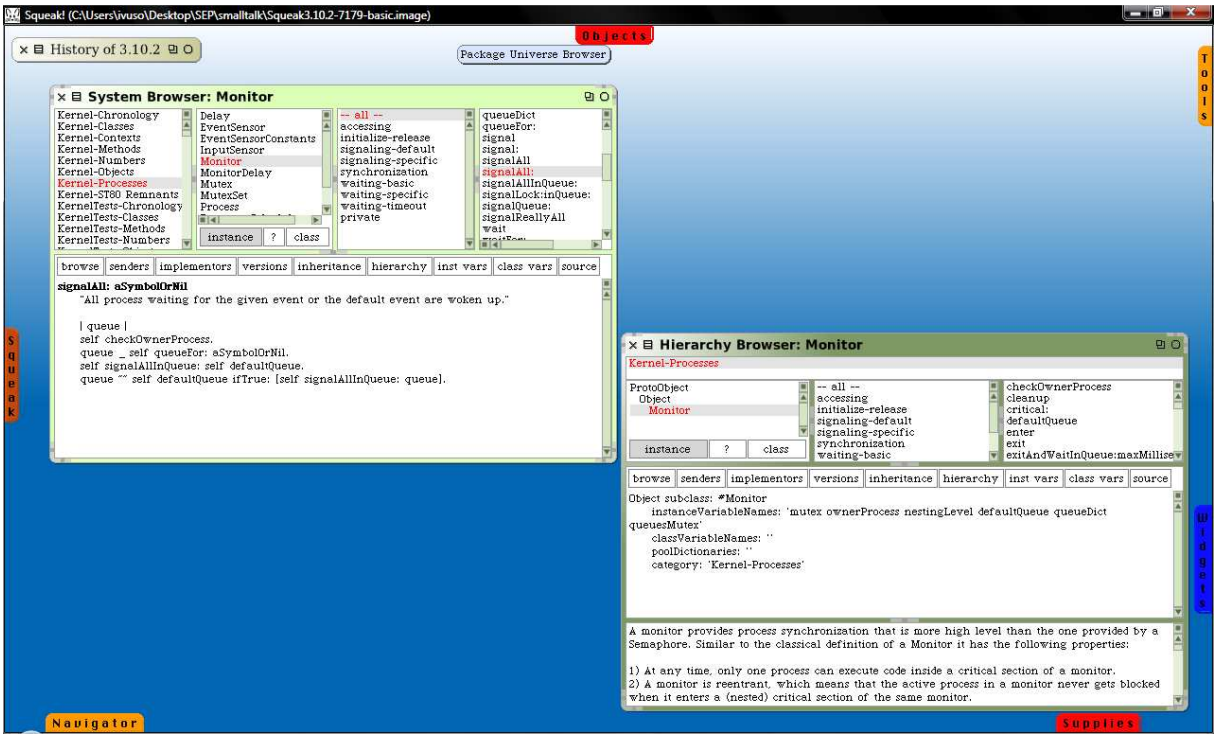
### 2.1.1 Smalltalk

Smalltalk je triedny objektovo orientovaný jazyk. Je to dynamický jazyk, v ktorom neexistuje rozdiel medzi dobou kompilácie a dobou spustenia. Smalltalk nie je tvorený len jazykom ale aj samotným vývojovým prostredím. V ňom má užívateľ k dispozícii niekoľko navzájom previazaných nástrojov. Typickým príkladom takého nástroja je System Browser, ktorý zobrazuje systém tried a metód. Triedy a ich metódy je možné v jazyku Smalltalk kategorizovať do jednotlivých kategórií. Pomocou tohto nástroja je tiež možné meniť zdrojové texty metód. Obrázok 2-1 obsahuje ukážku prostredia Squeak, čo je voľne dostupná implementácia jazyka Smalltalk. Nástroj System Browser má niekoľko variant. Ide napríklad o Hierarchy Browser, ktorý zobrazuje dedičnosť tried.

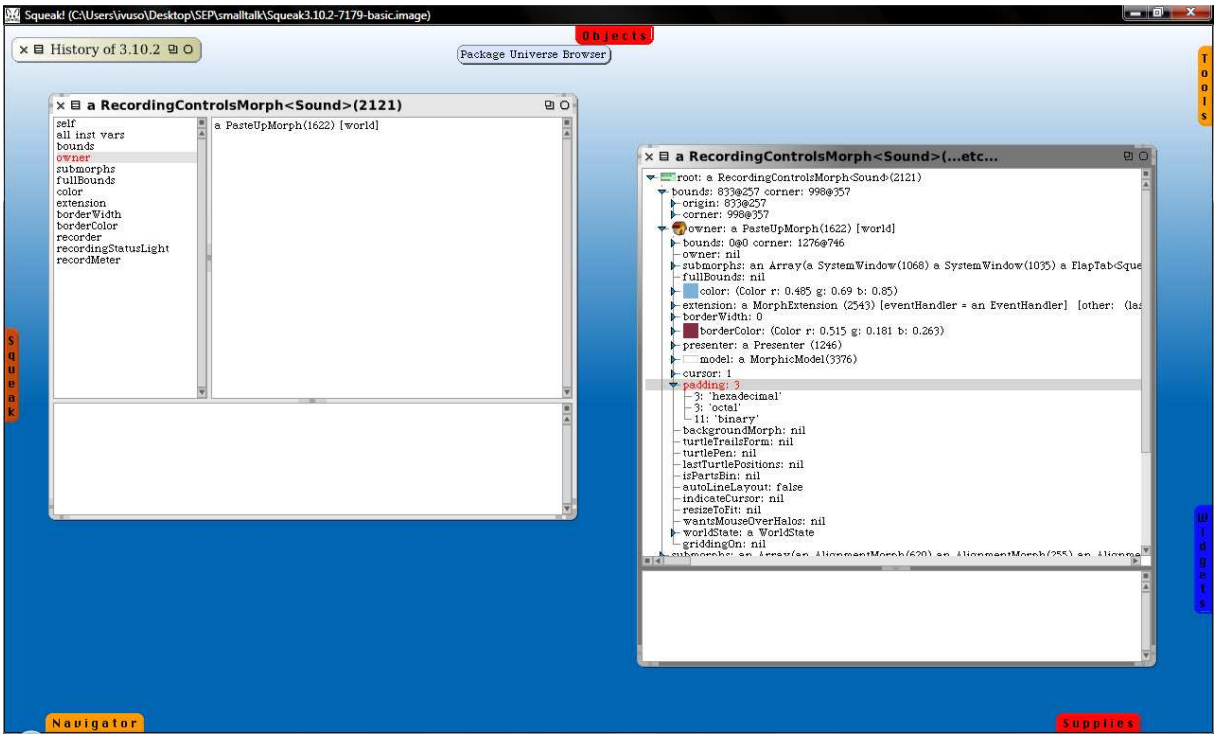
Ďalšími nástrojmi sú Inspector a Explorer. Sú zobrazené na obrázku 2-2. Nástroj Inspector zobrazuje hodnotu nejakého objektu a jeho členských premenných. Nástroj Explorer zobrazuje navyše aj hodnoty objektov, na ktoré ukazujú členské premenné tohto objektu. Pomocou prvého spomenutého nástroja je možné taktiež meniť hodnoty členských premenných.

Poslednými spomenutými nástrojmi sú Workspace a Transcript. Do okna nástroja Workspace je možné písať fragmenty kódu jazyka Smalltalk a okamžite ich vykonať. Umožňuje aj vytvoriť a ďalej používať dočasné premenné. Nástroj Transcript simuluje textový výstup. Obrázok 2-3 zobrazuje tieto dva nástroje.

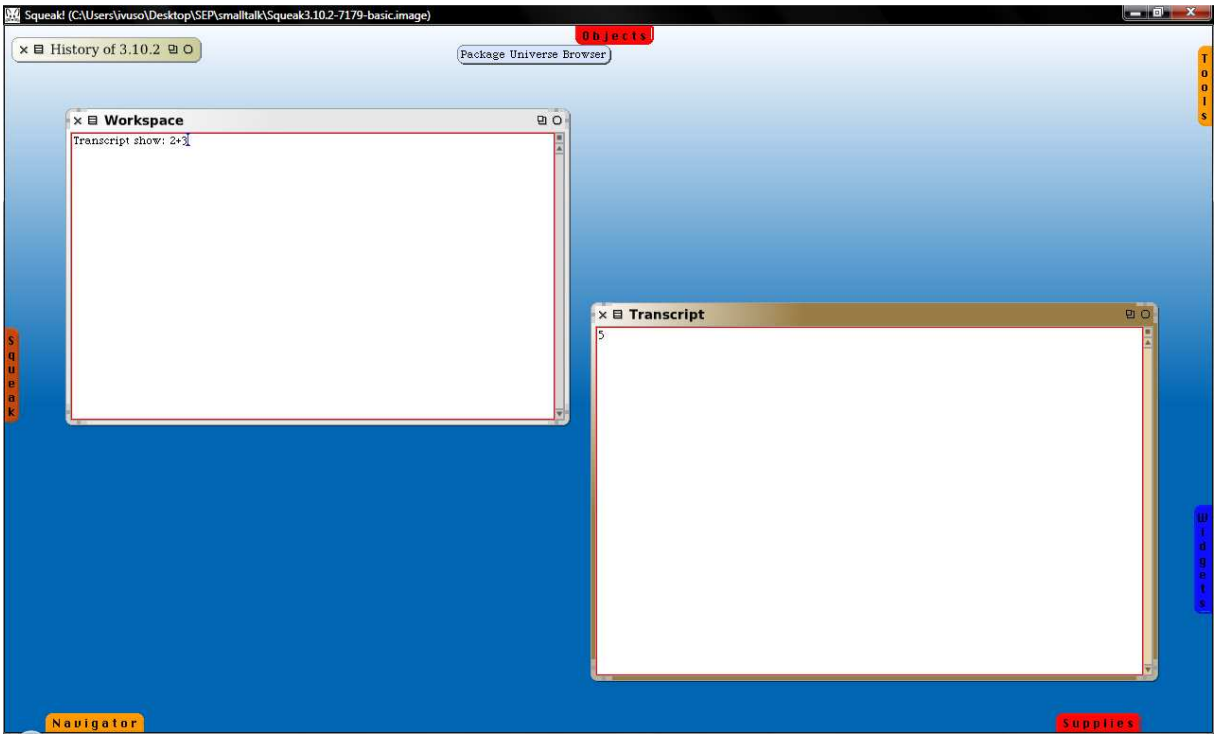
Ďalšie informácie o jazyku Smalltalk je možné nájsť napríklad v [2]. Viac o vzniku tohto jazyka je možné nájsť priamo od jeho autorov v [3].



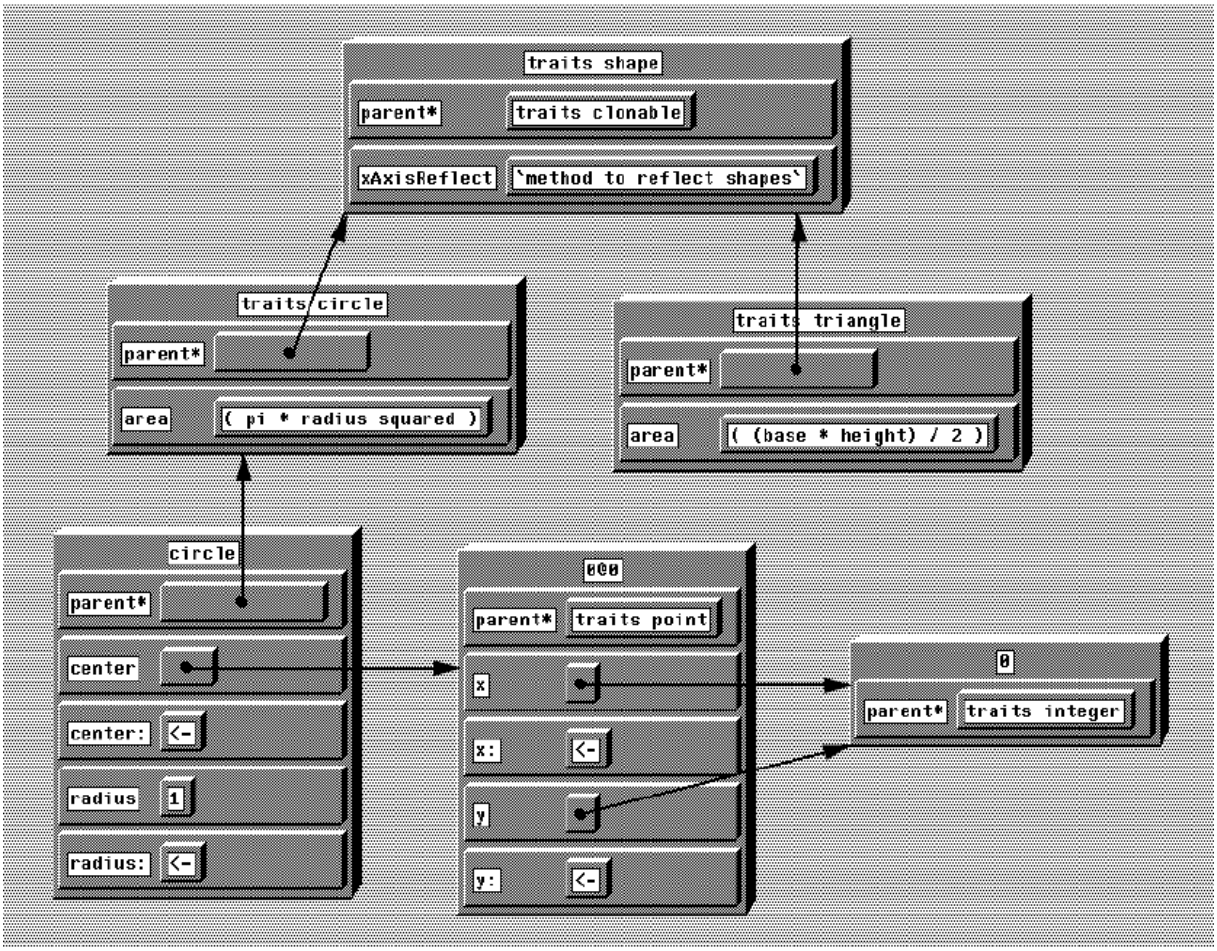

**Obrázok 2-1: System Browser a Hierarchy Browser**




**Obrázok 2-2: Inspector a Explorer**




**Obrázok 2-3: Workspace a Transcript**




**Obrázok 2-4: Self (obrázok prevzatý z [6])**

### 2.1.2 Self

Jazyk Self je podobne ako Smalltalk dynamickým objektovo orientovaným jazykom. Na rozdiel od neho je však prototypovo orientovaný. To znamená, že v ňom neexistuje dvojica konštrukcií trieda – objekt. Vlastnosti tried (napríklad dedičnosť) sa v tomto jazyku dosahujú vhodným posielaním správ. Podobne ako Smalltalk aj Self má vlastné vývojové prostredie, ktoré je vskutku unikátne. V ňom je možné každý objekt graficky zobraziť na ploche a manipulovať s ním. Referencie medzi objektmi sú zobrazené vo forme šípok. Ukážka tohto grafického systému sa nachádza na obrázku 2-4.

Ďalšie informácie o jazyku Self je možné nájsť napríklad v [4]. Viac o vzniku tohto jazyka je možné nájsť priamo od jeho autorov v [5]. Informácie o grafickom vývojom prostredí jazyka sa nachádzajú v [6]. Ďalšie informácie o jazykoch Smalltalk a Self sa nachádzajú v [7].

### 2.1.3 Visual Studio

Microsoft Visual Studio je integrované vývojové prostredie. Umožňuje vytvárať aplikácie na rôznych platformách a programovacích jazykoch. Na rozdiel od prostredí Smalltalk a Self sa tu primárne pracuje so zdrojovými súbormi (ktoré v prostrediach Smalltalk a Self priamo neexistujú). Napriek tomu však obsahuje niekoľko nástrojov, ktoré sú svojou funkčnosťou podobné tým, ktoré boli spomenuté v predchádzajúcich kapitolách.

Visual Studio obsahuje nástroj Class View, ktorý zobrazuje menné priestory, triedy a ich metódy. Nezobrazuje zdrojové texty metód, pretože tie sa nachádzajú v zdrojových súboroch. Kliknutím na metódu v nástroji Class View je však možné ku jej zdrojovým textom prejsť. Ukážka nástroja Class View sa nachádza na obrázku 2-5.

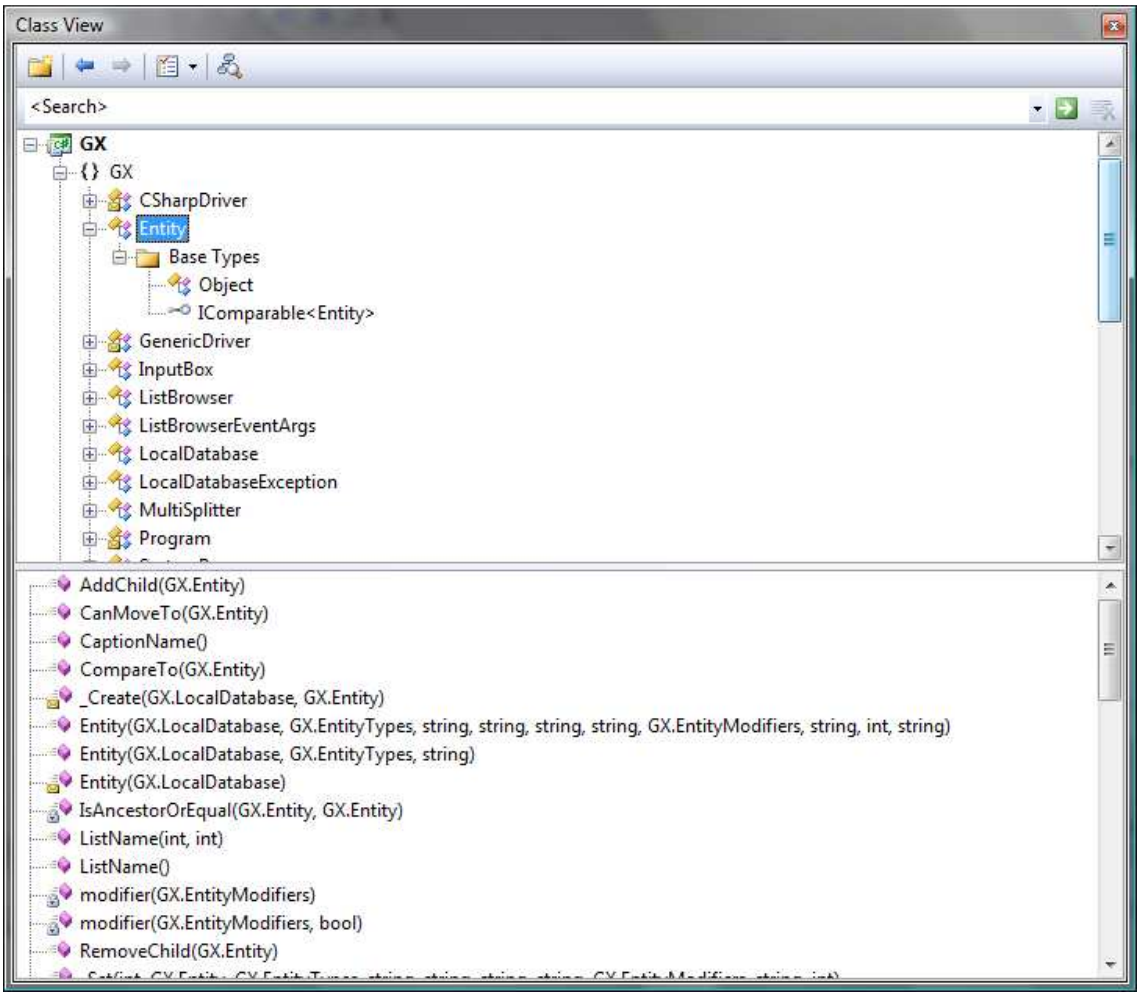
Ďalším nástrojom je Immediate Window, ktorý plní podobnú úlohu ako Workspace v jazyku Smalltalk. Aj pri kompilovaných jazykoch ako napríklad C# je možné do neho napísať fragmenty kódu a vykonať ich. Tento nástroj je možné dokonca použiť aj pri jazykoch C/C++. Ukážka nástroja sa nachádza na obrázku 2-6.

Okrem toho obsahuje Visual Studio aj nástroj Object Test Bench. Pomocou neho je možné vytvárať inštancie tried v čase, kedy aplikácia nie je ani spustená, čo ešte viac znižuje rozdiel medzi dobou kompilácie a spustenia v programoch na platforme .NET.

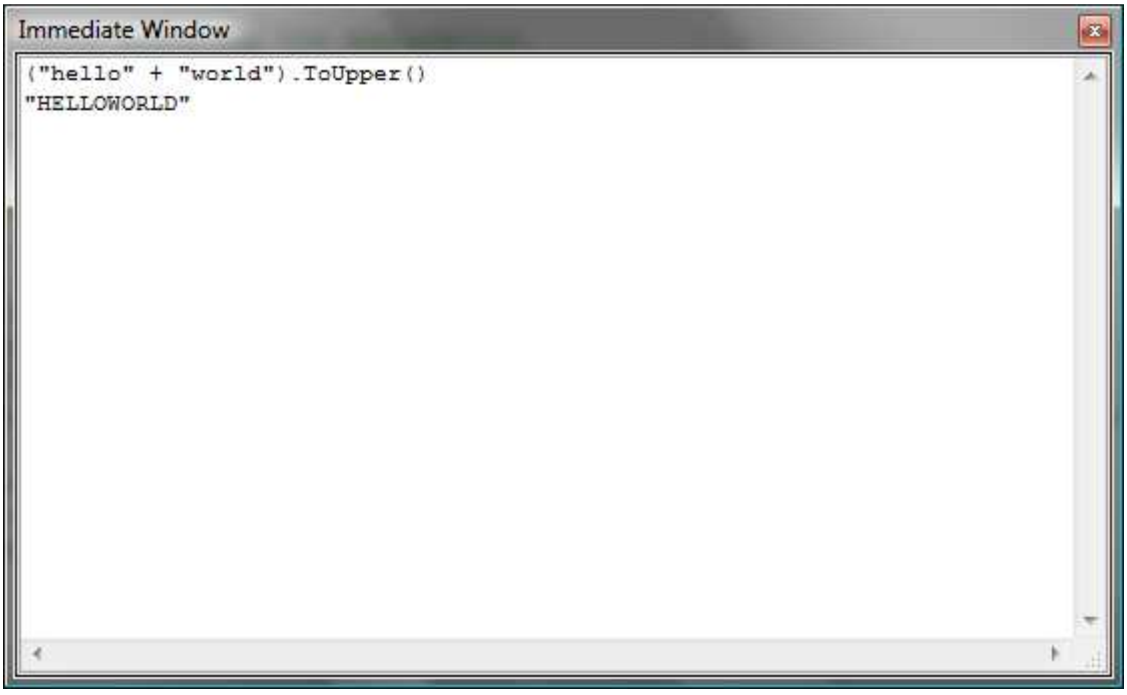
### 2.1.4 Code Bubbles

Code Bubbles je nástroj určený pre jazyk Java, ktorý sa svojím vzhľadom podobá na prostredie jazyka Self. Užívateľ vidí plochu, na ktorú si môže postupne pridávať fragmenty programu (triedy, členské premenné, metódy, komentáre, rôzne značky a pod.). Tieto fragmenty nazýva bublinami. S bublinami je potom možné ľubovoľne manipulovať. Viac informácií sa nachádza v [8].

Nástroj je samostatným vývojovým prostredím. Obsahuje niekoľko zaujímavých funkcií. Ide napríklad o určenie najkratšej cesty (v grafe objektov a referencií) medzi dvomi objektmi.

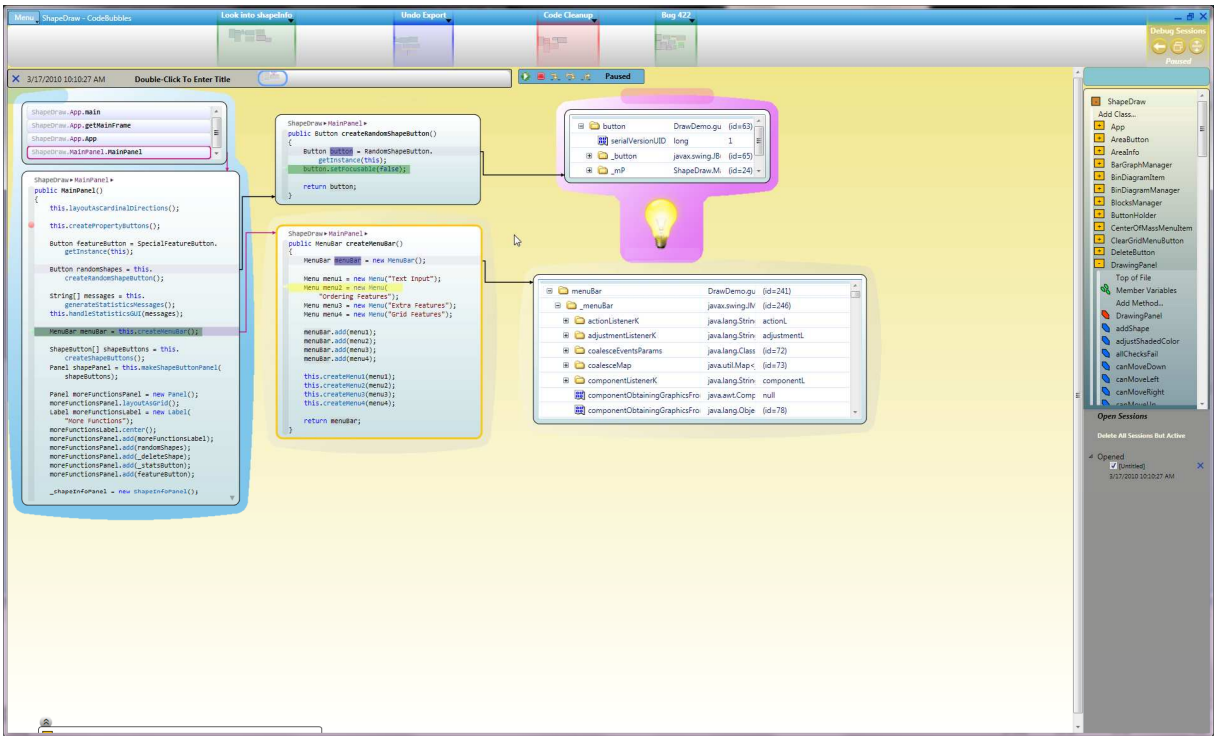



**Obrázok 2-5: Class View**




**Obrázok 2-6: Immediate Window**

Ukážka prostredia Code Bubbles sa nachádza na obrázku 2-7.



**Obrázok 2-7: Code Bubbles**

# 2.2 Výber nástroja

Cieľom tejto práce je navrhnúť a implementovať niektorý z nástrojov spomenutých v predchádzajúcej kapitole. Preto bolo potrebné pri výbere zodpovedať nasledujúce otázky:

1. O aký nástroj konkrétne pôjde ?
2. Bude nástroj využiteľný v dobe kompilácie alebo v dobe spustenia ?
3. Ktoré programovacie jazyky a platformy bude nástroj podporovať ?
4. Pôjde o samostatný nástroj, alebo bude existovať vo forme rozšírenia existujúceho vývojového prostredia ? O ktoré vývojové prostredie pôjde ?
5. Aké prostriedky sa použijú na jeho vytvorenie ?

Nástroj by mal byť v každom prípade rozšírením existujúceho vývojového prostredia. Je to možnosť, ktorá je ohľaduplnejšia k zvyklostiam potenciálnych užívateľov. Navyše užívatelia využívajú pri svojej činnosti niekoľko nástrojov naraz. Preto by nebolo veľmi rozumné ich o túto možnosť pripraviť. Vývojové prostredie Visual Studio je v operačnom systéme Microsoft Windows hojne používané a obľúbené medzi vývojármi a preto nástroj rozšíri práve toto prostredie.

Mal by podporovať jazyk C# a/alebo C/C++, keďže ide o rozšírené jazyky. S tým súvisí platforma .NET alebo x86 (poprípade jej 64 bitové rozšírenie).

Visual Studio obsahuje varianty prakticky všetkých nástrojov spomenutých v kapitole 2.1.1 venovanej jazyku Smalltalk. Preto bola pozornosť zameraná na jazyk Self spomenutý v kapitole 2.1.2. Bude vytvorený dynamický nástroj pripomínajúci prostredie jazyka Self (obrázok 2-4). Jeho meno bude Object Viewer.

Celkovo bude teda nástroj slúžiť na manipuláciu objektov programu v dobe spustenia. Bude podporovať programy napísané v jazykoch C/C++ na 32 a 64 bitovej platforme v operačnom systéme Windows. Primárne bude existovať vo forme rozšírenia prostredia Visual Studio. Nasledujúce kapitoly popisujú niektoré prostriedky použiteľné pri jeho vývoji.

## 2.3 Kontext procesu

Inštancia programu, ktorý je obyčajne reprezentovaný vo forme spustiteľného súboru sa nazýva proces. Každý proces má k dispozícii súvislý virtuálny adresný priestor. Ten obsahuje všetky globálne (statické premenné), lokálne (premenné v procedúrach) a dynamické (premenné na hromade) objekty programu spolu so štruktúrami využívanými operačným systémom. Všetky tieto dáta tvoria *kontext procesu* (jeho stav). Kontext procesu sa skladá z niekoľkých častí, ktoré sú popísané v nasledujúcich podkapitolách. Viac o vlastnostiach procesov je možné sa dozvedieť napríklad v [9].

### 2.3.1 Moduly

Každý proces obsahuje niekoľko načítaných spustiteľných súborov – modulov. Obyčajne ide o pôvodný spustiteľný súbor, z ktorého vznikol proces a niekoľko dynamických knižníc. Každý modul obsahuje kód a statické premenné zdieľané všetkými vláknami. Jednotlivé moduly sa nachádzajú na samostatných rozsahoch adries v adresnom priestore. Proces typicky pridáva a odoberá moduly v pamäti podľa potreby.

### 2.3.2 Vlákna

Každý proces obsahuje aspoň jedno vlákno. Každé vlákno má pridelený samostatný zásobník, ktorý obsahuje všetky lokálne premenné a parametre a je tvorený postupnosťou volaných procedúr. Každý zásobník sa nachádza na samostatnom rozsahu adries v adresnom priestore. Podobne ako moduly aj vlákna dynamicky vznikajú a zanikajú počas existencie procesu. Každé vlákno má oddelený kontext registrov.

Jednotlivé vlákna majú priradené vlastné inštancie priestoru TLS (Thread Local Storage). Premenná nachádzajúca sa v tomto priestore sa správa podobne ako klasická statická premenná. Je určená identifikátorom. Jediným rozdielom je, že každé vlákno má priradenú vlastnú inštanciu tejto premennej.

### 2.3.3 Dynamická pamäť

Okrem modulov a vlákien sa v adresnom priestore nachádzajú ďalšie dynamicky alokované bloky. Ide o bloky pamäte, ktoré sa vytvárajú napríklad pri operácií new na hromade. Tieto môžu vznikať a zanikať a sú úplne nezávislé od modulov a vlákien.

## 2.4 Pamäť objektov

Čisto objektovo orientované jazyky obsahujú pamäť objektov, kde každý objekt má jednoznačnú identifikáciu – adresu objektu. Jednotlivé objekty obsahujú vo svojich členských premenných referencie na ďalšie objekty. Tento vzťah referencie je jediným možným vzťahom medzi objektmi.

Niektoré jazyky toto pravidlo kvôli zvýšeniu rýchlosti porušujú tým, že obsahujú hodnotové typy. Objekty hodnotových typov sa nachádzajú v iných materských objektoch a taktiež nemajú svoju jednoznačnú identifikáciu. To znamená, že žiadna premenná nemôže obsahovať referenciu na takéto objekty a tie sú vždy vlastnené maximálne jedným objektom. Ide napríklad o jazyk C#.

Jazyky C a C++ idú v tomto ešte ďalej. Na jednoznačnú identifikáciu objektov v týchto jazykoch je ich adresa nepostačujúca. Na obrázku 2-8 sa nachádza krátky úsek programu. Po vytvorení premennej `u` nastane zaujímavá situácia, kedy sa jednotlivé objekty `u`, `c1`, `c2`, `a`, `b` budú nachádzať na tej istej adrese[1]. Je to tak preto, pretože takéto jednoduché typy nepotrebujú mať v jazykoch C a C++ žiadnu réžiu. Viac o týchto skutočnostiach je možné sa dozvedieť v [10].

```
class C1
{
    int a;
};

class C2
{
    int b;
    int c;
};

union U
{
    C1 c1;
    C2 c2;
};

U u;
```

**Obrázok 2-8: Typy v C/C++**

[1] Je to pravdepodobné, ale v každom prípade závislé na implementácií.

Taktiež je možné ukazovať referenciou „dovnútra“ objektu Je úplne bežné, ak existuje ukazovateľ na objekty `a`, `b`, `c`, poprípade `c1`, `c2`.

Čo je ešte horšie, vďaka existencii typu `union` neplatí, že sa každý objekt nachádza maximálne v jednom materskom objekte. Premenné `a` a `b` obsahujú v podstate ten istý objekt a napriek tomu sa nachádzajú v rôznych materských objektoch `c1` a `c2`.

Prítomnosť manuálnej správy pamäte ešte viac komplikuje situáciu. Pri dynamicky alokovaných objektoch je nemožné určiť ich životnosť. Na danej adrese sa počas behu programu môže vystriedať niekoľko rôznych typov objektov alebo dokonca daná adresa môže byť súčasťou iného objektu.

Keďže adresa objektu neidentifikuje jednoznačne objekt v jazykoch C a C++ je potrebné nájsť inú identifikáciu. Tou môže byť kombinácia adresy a typu. Tak je možné odlíšiť objekty `c1` a `c2` z predchádzajúceho príkladu. Napriek rovnakým adresám majú totiž rôzny typ. Premenné `a` a `b` naďalej obsahujú ten istý objekt, čo je ale úplne normálne.

## 2.5 Rozhranie CodeModel

Vývojové prostredie Microsoft Visual Studio umožňuje vytváranie ľubovoľných rozšírení. Tie majú prístup ku veľkému počtu rozhraní, ktoré sú poskytované prostredím. Pomocou nich je možné do detailov manipulovať s vývojovým prostredím. Existujú aj rozhrania, pomocou ktorých je možné pristupovať ku kódovému modelu projektu [11]. Ide vlastne o model zdrojových textov vo forme abstraktného syntaktického stromu. Tak je možné napríklad zistiť zoznam tried, ich členských premenných a metód a ich parametrov v aktuálnom projekte.

Vstupnými bodmi do týchto štruktúr sú rozhrania `CodeModel` a `FileCodeModel`. Pomocou nich je možné pristúpiť ku kódovému modelu celého projektu alebo len jediného zdrojového súboru. Jednotlivé programové elementy sú reprezentované rozhraním `CodeElement`. To obsahuje vlastnosti ako názov elementu, typ a jeho poloha v zdrojových textoch. Niektoré elementy obsahujú v sebe ďalšie elementy (napr. metóda obsahuje zoznam parametrov). Tento zoznam je reprezentovaný rozhraním `CodeElements`. Pre niektoré typy elementov (napr. pre triedu) existujú samostatné rozhrania odvodené z `CodeElement` (napr. `CodeClass` pre triedu). Názov všetkých týchto rozhraní sa začína reťazcom `Code`.

Tieto rozhrania nie sú špecifické len pre jeden jazyk. Sú použiteľné ako pre C#, tak aj pre C++ a iné. Do modelu nie je zahrnutý kód metód ani iné pre jazyk špecifické vlastnosti.

Pre jednotlivé programovacie jazyky môže prostredie obsahovať ďalšie špecifické, odvodené rozhrania. Príkladom je jazyk C++. Prostredie obsahuje sadu rozhraní, ktorých názov sa začína reťazcom `VCCode` (napr. rozhranie `VCCodeMacro` popisujúce makro v jazykoch C/C++).

Nie všetky vlastnosti týchto rozhraní sú vždy platné. Pri pokuse o prístup k takým vlastnostiam je vygenerovaná výnimka. Niektoré vlastnosti je možné aj meniť.

Prostredie generuje udalosti o zmene kódového modelu. Na ich príjem sa používa rozhranie `CodeModelEvents`. To obsahuje 3 udalosti, ktoré sa generujú pri vytvorení nového elementu, zmene vlastností existujúceho elementu a zrušení elementu. Tieto udalosti sú generované s miernym časovým oneskorením tak ako užívateľ upravuje zdrojové texty.

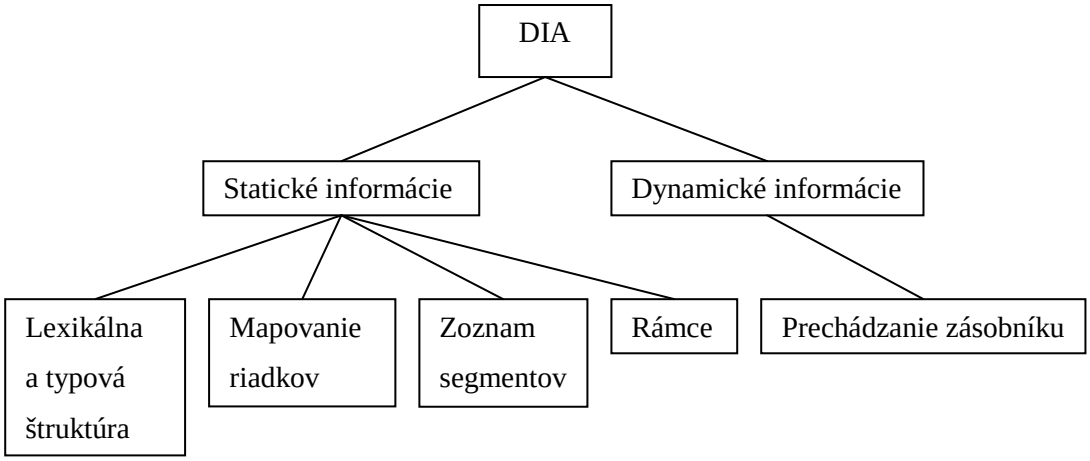
## 2.6 Rozhranie Debug Interface Access

Pri kompilácií zdrojových textov dochádza k ich prevodu do cieľovej podoby, ktorá je často postavená na iných princípoch ako zdrojové texty. Časť informácie, ktorá nie je potrebná pre beh výsledného programu sa nevyhnutne stráca. V jazykoch C/C++ ide napríklad o názvy identifikátorov, užívateľsky definované typy, argumenty a lokálne premenné funkcií, zoznam zdrojových súborov a obsah riadkov v týchto súboroch. Obsah riadkov sa stráca napríklad aj pri kompilovaní v prostredí .NET. V jazyku Smalltalk sa uchováva skompilovaná podoba metód oddelene od ich zdrojovej podoby.

Ladiace nástroje preto potrebujú pre plnohodnotnú činnosť okrem skompilovaného programu aj zdrojové texty a symbolické informácie. Symbolické informácie obsahujú mapovanie medzi zdrojovou a skompilovanou podobou programu a sú výsledkom kompilácie. Pomocou nich je možné napríklad určiť výslednú virtuálnu adresu zdrojového riadku alebo adresu lokálnej premennej.

Rozhranie Debug Interface Access [12] (ďalej už len DIA) umožňuje prístup k takýmto symbolickým informáciám, ktoré sú generované prekladačmi firmy Microsoft. Ich podoba sa môže líšiť s verziou prekladača. V dobe písania tejto práce sú symbolické informácie ukladané oddelene v súboroch s príponou pdb (skratka z anglického *Program Database*).

DIA obsahuje kolekciu COM rozhraní. Ich názvy sa vždy začínajú reťazcom `IDia`. Je možné ich rozdeliť do piatich skupín ako na obrázku 2-9. Štyri skupiny obsahujú statické informácie (informácie priamo generované kompilátorom). Zostávajúca skupina obsahuje dynamické informácie (informácie zo spusteného programu). V nasledujúcich podkapitolách sú tieto skupiny rozhraní popísané.




**Obrázok 2-9: Štruktúra rozhrania DIA**

### 2.6.1 Lexikálna a typová štruktúra

Táto skupina obsahuje hlavné symbolické informácie. Obsahuje informácie o moduloch (súbory obj, ktoré vstupujú do linkera), o funkciách a metódach, o blokoch v nich, o globálnych, lokálnych a členských premenných, o parametroch funkcií, o štruktúre užívateľsky definovaných typov a pod. Informácie sú uložené vo forme pripomínajúcej orientovaný graf. Uzol grafu sa nazýva symbol a je reprezentovaný rozhraním `IDiaSymbol`. Pre každý symbol platia nasledujúce pravidlá:

1. Každý symbol má svoje jedinečné identifikačné číslo.
2. Každý symbol má svoj druh.
3. Každý symbol okrem koreňového má svojho lexikálneho predchodcu.
4. Každý symbol môže mať triedneho predchodcu.
5. Každý symbol môže mať následníkov.
6. Každý symbol môže mať ďalšie odkazy na iné symboly.
7. Každý symbol môže mať ďalšie vlastnosti.

V tabuľkách 2-1 a 2-2 sú uvedené všetky možné druhy symbolov spolu s krátkym popisom. Druh symbolu môže byť buď lexikálneho (popisuje štruktúru súborov), alebo triedneho (popisuje typové informácie) charakteru.

| **Druh symbolu** | **Popis** |
|---|---|
| `SymTagExe` | koreňový symbol obsahujúci základné informácie o programe |
| `SymTagCompiland` | popisuje modul pri preklade (moduly vstupujú do linkera) |
| `SymTagCompilandDetails` | obsahuje ďalšie informácie o module |
| `SymTagCompilandEnv` | obsahuje ďalšie informácie o module |
| `SymTagFunction` | popisuje funkciu (alebo metódu) |
| `SymTagBlock` | popisuje blok vo funkcií (text ohraničený {}) |
| `SymTagThunk` | popisuje linkerom generovaný krátky kód, ktorý volá skutočný kód |
| `SymTagFuncDebugStart` | určuje miesto vo funkcií, kde sa začína užitočný kód (telo) |
| `SymTagFuncDebugEnd` | určuje miesto vo funkcií, kde sa končí užitočný kód (telo) |
| `SymTagLabel` | popisuje nejaké významné miesto v kóde |
| `SymTagData` | popisuje globálnu, lokálnu alebo členskú premennú |
| `SymTagPublicSymbol` | obsahuje symbol, ktorý je verejný |
| `SymTagUsingNamespace` | popisuje konštrukciu `using namespace` |
| `SymTagAnnotation` | popisuje nejaké významné miesto v kóde |
| `SymTagCustom` | obsahuje iné špeciálne informácie |

**Tabuľka 2-1: Lexikálne druhy symbolov**

Základnú myšlienku ukazuje nasledujúci jednoduchý príklad. Obrázok 2-10 ukazuje krátky úsek kódu v jazyku C++. Príklad obsahuje metódu `drive`. Metóda `drive` je reprezentovaná symbolom s druhom `SymTagFunction`. Ako lexikálneho predchodcu obsahuje referenciu na symbol modulu (`SymTagCompiland`), kde je definovaná. Ako triedneho predchodcu obsahuje referenciu na symbol triedy `Vehicle` (`SymTagUDT`). Ďalej obsahuje referenciu na symbol typu metódy (`SymTagFunctionType`), ktorý následne obsahuje symboly typov jednotlivých parametrov (`SymTagFunctionArgType`) vo forme následníkov. V tomto prípade symbol typu parametra obsahuje referenciu na symbol typu ukazovateľa (`SymTagPointerType`) a ten konečne obsahuje referenciu na symbol triedy `Velocity` (`SymTagUDT`). Je možné tiež zistiť členov triedy `Vehicle` ako následníkov jej symbolu. V tomto prípade má trieda `Vehicle` dvoch následníkov: symbol s druhom `SymTagFunction` a symbol s druhom `SymTagData`.

| Druh symbolu | Popis |
|---|---|
| SymTagUDT | popisuje dátový typ definovaný pomocou `class`, `struct`, alebo `union` |
| SymTagEnum | popisuje dátový typ definovaný pomocou `enum` |
| SymTagBaseClass | obsahuje referenciu na bázovú triedu pri dedičnosti |
| SymTagFunctionType | reprezentuje typ funkcie (kombinácia návratového typu a typu parametrov) |
| SymTagFunctionArgType | obsahuje referenciu na dátový typ parametra |
| SymTagPointerType | popisuje ukazovateľ alebo referenciu |
| SymTagArrayType | popisuje pole |
| SymTagDimension | obsahuje rozsah poľa |
| SymTagTypedef | popisuje konštrukciu `typedef` |
| SymTagFriend | popisuje konštrukciu `friend` |
| SymTagVTable | obsahuje informácie o virtuálnej tabuľke |
| SymTagVTableShape | obsahuje informácie o virtuálnej tabuľke |
| SymTagBaseType | popisuje skalárny dátový typ |
| SymTagManagedType | obsahuje špeciálne informácie |
| SymTagCustomType | obsahuje iné špeciálne informácie |

**Tabuľka 2-2: Triedne druhy symbolov**

```
class Vehicle
{
    void drive(Velocity * vel)
    {
    }
    int speed;
};
```

**Obrázok 2-10: Príklad symbolov**

Ako bolo povedané, všetky symboly sú reprezentované jedným rozhraním `IDiaSymbol`. To obsahuje ďalšie vlastnosti, ktorých platnosť je závislá na druhu symbolu. Z druhu symbolu teda vyplýva, ktoré vlastnosti je možné získať. Medzi ďalšie vlastnosti patrí napríklad názov, virtuálna adresa, adresa na zásobníku, adresa relatívne k `this` a bajtová šírka symbolu. Tieto informácie sú bližšie popísané v [12].

Pri analýze rozhrania DIA bolo ďalej empiricky zistených niekoľko skutočností:

1. Databáza obsahuje duplicitné symboly, t.j. symboly ktoré majú rovnaký obsah, ale rôzne identifikačné číslo.

2. Databáza obsahuje symboly, ktoré sú rekurzívnym prechodom grafu symbolov nedostupné. Je možné sa k nim dostať len cez ďalšie odkazy v symboloch.
3. Databáza obsahuje staré symboly z predchádzajúcich verzií zdrojových textov, čo vyplýva z faktu, že kompilátor pracuje inkrementálne.

Okrem samotnej databázy symbolov obsahuje DIA aj niekoľko operácií na nájdenie symbolu podľa názvu, adresy a pod.

### 2.6.2 Mapovanie riadkov

Táto skupina obsahuje rozhrania, ktoré definujú vzťah medzi zdrojovým textom a cieľovým binárnym kódom. Rozhranie `IDiaLineNumber` popisuje:

1. miesto (riadok a stĺpec) začiatku a miesto (riadok a stĺpec) konca príkazu alebo výrazu zdrojového súboru
2. virtuálnu adresu a dĺžku cieľového kódu, ktorý bol vytvorený z daného príkazu alebo výrazu

Túto informáciu využívajú ladiace nástroje napríklad na to, aby bolo možné krokovať program po riadkoch a nie po jednotlivých inštrukciách.

### 2.6.3 Zoznam segmentov

Táto skupina obsahuje rozhrania, ktoré popisujú segmenty programu. Typický program je rozdelený do niekoľkých segmentov. Obsahuje segment obsahujúci kód, segment obsahujúci konštantné dáta, segment obsahujúci nekonštantné dáta, segment neinicializovaných dát a pod. Toto popisuje rozhranie `IDiaSegment`. Táto informácia môže byť použiteľná pri prechádzaní zásobníka, kedy je potrebné rozhodnúť či daná virtuálna adresa obsahuje spustiteľný kód alebo nie.

### 2.6.4 Rámce

Táto skupina obsahuje rozhrania, ktorých informácie sa používajú pri prechádzaní zásobníka. Za normálnych okolností prekladače generujú kód funkcií tak, že je možné jednoduchým prechádzaním linkovaného zoznamu prechádzať všetky rámce funkcií na zásobníku a tak určiť jednotlivé volané funkcie [13]. Prekladače však môžu v špeciálnych prípadoch kód funkcie urýchliť vynechaním ukazovateľa na rámec (angl. *frame pointer omission*). Chýbajúca informácia je potom zachovaná pomocou rozhrania `IDiaFrameData`. Toto rozhranie obsahuje vlastnosti ako veľkosť bloku lokálnych premenných a parametrov pre takú funkciu.

### 2.6.5 Prechádzanie zásobníka

Zatiaľ čo predchádzajúce skupiny rozhraní umožňovali prístup ku statickým informáciám konkrétneho programu, rozhrania patriace do skupiny prechádzanie zásobníka umožňujú určiť zoznam a vlastnosti aktívnych funkcií na existujúcom zásobníku (angl. *stack walking*).

Rozhrania patriace do tejto skupiny obsahujú len algoritmy. Obsah konkrétneho zásobníka, pamäte a registrov sa musí dodať explicitne. To isté platí pre statické symbolické informácie spomenuté v prechádzajúcich kapitolách. Tie sa musia dodať explicitne tiež. V praxi obsahuje jeden proces niekoľko programov, typicky jeden spustiteľný súbor a niekoľko dynamických knižníc. Na zásobníku preto môžu existovať aktívne funkcie z rôznych programov.

Zásobník sa prechádza pomocou rozhrania `IDiaEnumStackFrames`. Iteratívnym volaním jeho metódy `Next` sa získava referencia na rozhranie `IDiaStackFrame`. To už popisuje jednu konkrétnu inštanciu funkcie na zásobníku. Ako prvá sa vždy vráti aktuálna funkcia. Vstupný bod celého procesu sa vráti ako posledný.

Prechádzanie zásobníka je závislé na implementácií rozhrania `IDiaStackWalkHelper` klientom. Toto rozhranie obsahuje metódy na určenie obsahu pamäte na danej virtuálnej adrese, na určenie obsahu registrov, určenie symbolu podľa danej adresy a pod. Každá iterácia `Next` zároveň na konci zmení obsah pomyslených registrov tak, aby sa zmenila aktuálna funkcia na ďalšiu v poradí.

Zmyslom oddelenia algoritmu a procesu získavania a manipulácie dát je v tomto prípade to, aby bolo možné prechádzať zásobník nielen v živých procesoch, ale napríklad aj v ich snímkoch na disku (angl. *memory dump*).

### 2.6.6 Zhrnutie rozhrania DIA

Rozhranie DIA je v dobe písania tejto práce používané vývojovými nástrojmi firmy Microsoft. Väčšinou je nad ním ešte vytvorená nejaká forma abstrakcie. DIA používa štandard COM, je preto použiteľná ako v jazyku C++ tak aj v jazykoch na platforme .NET. Bežne sa dodáva s inštaláciou produktu Microsoft Visual Studio. Je uložená vo forme dynamickej knižnice (súbor dll).

Dokumentácia k jednotlivým rozhraniam je pomerne strohá. Súčasťou je iba jedna ukážková aplikácia Dia2dump.

# 3 Návrh

## 3.1 Neformálna špecifikácia

Cieľom je vytvoriť nástroj Object Viewer, ktorý bude slúžiť na podporu vývoja aplikácií v čase spustenia, t.j. pri ladení a testovaní. Bolo by ho možné použiť pri manipulácií s komplikovanými dátovými štruktúrami ale aj ako výukový nástroj. Primárne bude pracovať s aplikáciami napísanými v jazykoch C/C++ na 32 a 64 bitových platformách.

Nástroj má pripomínať prostredie jazyka Self (obrázok 2-4). Bude vizuálne zobrazovať obsah pamäte ladeného procesu. Jednotlivé živé objekty budú zobrazené ako objekty na virtuálnej ploche. Vo svojom vnútri budú obsahovať zoznam členských premenných, ich typy a hodnoty. Medzi objektmi, ktoré na seba ukazujú budú zobrazené orientované hrany. V konečnom dôsledku bude nástroj vlastne zobrazovať podmnožinu grafu objektov.

Na obrázku 3-1 je načrtnutá možná grafická podoba jedného objektu pre ilustráciu. Ide o objekt triedy `Dog` na virtuálnej adrese `0x00402020`. Jeho trieda obsahuje 4 členské premenné: názov, váhu, farbu a ukazovateľ na iný objekt. Samotný objekt môže byť rozbalený podobne ako na obrázku, alebo môže byť zbalený. V tom prípade by jeho členské premenné nebolo vidno. To by zvýšilo prehľadnosť plochy.

Užívateľsky definované typy obsahujú členské premenné, ktoré môžu byť sami o sebe užívateľsky definovanými typmi. Preto môžu objekty v sebe obsahovať vnorené objekty. Tie bude možné ďalej rozbaliť. Obrázok 3-2 ukazuje rozbalenú členskú premennú `color`.

Niektoré premenné ukazujú na iné objekty. Ide o ukazovatele, referencie, ale vo svojej podstate aj o iterátory. Podobne ako pri rozbaľovaní vnorených objektov nie je nutné implicitne tento vzťah zobraziť, ale počkať až na pokyn od užívateľa. Obrázok 3-3 ukazuje takú situáciu. Objekt typu `Dog` ukazuje pomocou svojej členskej premennej `owner` na iný objekt typu `Person`.

Na obrázkoch je ďalej vidno, že aj keď je typ reťazca `std::string` užívateľsky definovaným typom a jeho hodnota je obsiahnutá až v hodnotách jeho členských premenných, je predsa len zobrazený obsah reťazca. Väčšinou totiž pri objektoch takýchto elementárnych typov platí, že nie je dôležitý obsah ich členských premenných ale samotná hodnota, ktorú reprezentujú. Ďalším príkladom je typ `Color`, pri ktorom je možné zobraziť obsah jednotlivých farebných zložiek v ľubovoľnom farebnom modeli a to nezávisle na štruktúre tohto typu. Dokonca je možné pre známe farby prezentovať aj ich názov.

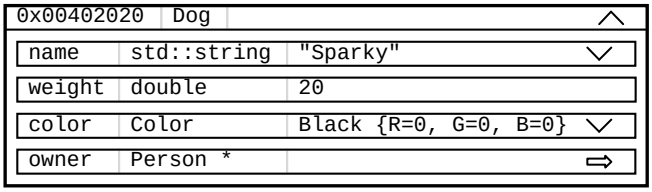
Nástroj na žiadosť zobrazí zoznam globálnych premenných, zoznam vlákien a zoznam všetkých typov a užívateľ si ich bude môcť pridať na plochu. Pri zozname typov ešte užívateľ explicitne zadá adresu a tak bude môcť zobraziť objekt ľubovoľného typu na ľubovoľnej adrese. Z týchto zá-

kladných objektov sa užívateľ bude môcť dostať na zvyšné objekty postupným rozbaľovaním vnorených objektov a sledovaním ukazovateľov. Objekty bude možné presúvať na virtuálnej ploche a tiež nepotrebné objekty skryť. Objekty, na ktoré už na ploche neexistuje žiadna referencia a užívateľ ich explicitne nezobrazil môže nástroj skryť automaticky.

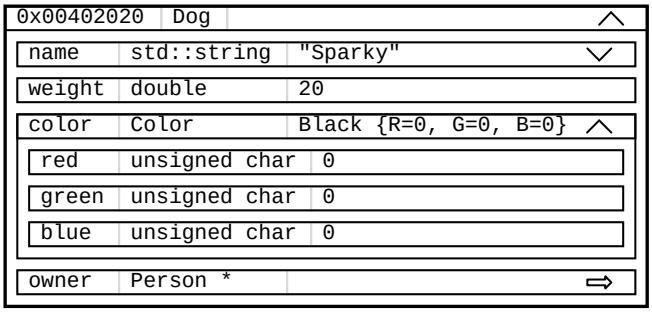
Z jednotlivých objektov bude užívateľ môcť prejsť na definíciu ich tried a premenných v zdrojových textoch. Nástroj bude môcť zmeniť hodnotu objektu a umožní vykonanie niektorej z jeho metód. Žiadanou vlastnosťou je aj možnosť priblížiť a oddialiť pohľad a nájdenie všetkých odkazov na vybraný objekt.

Stav plochy by mal byť čo možno najviac imúnny voči krokovaniu programu. To znamená napríklad to, že objekty, ktoré boli rozbalené by mali zostať rozbalenými aj po spustení a opätovnom pozastavení behu procesu. Pri spustenom stave by navyše mala byť plocha zablokovaná.

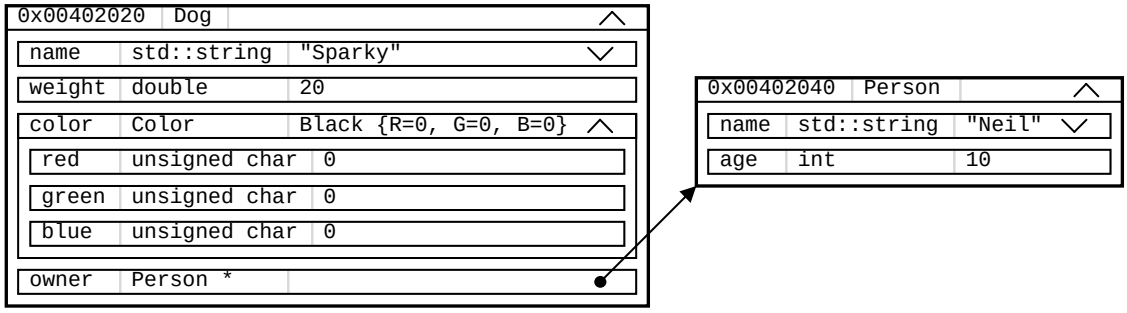
Nástroj bude existovať vo forme samostatnej aplikácie ale zároveň aj ako rozšírenie vývojového prostredia Visual Studio. Bude umožňovať k nemu pripojiť vlastné rozšírenia, ktoré by rozširovali jeho schopnosť prívetivo zobraziť obsah objektov aj ďalších (vlastných) užívateľských typov podobne ako je to pri `std::string`.




**Obrázok 3-1: Objekt**




**Obrázok 3-2: Rozbalená premenná**




**Obrázok 3-3: Ukazovateľ**

## 3.2 Definícia pojmov

Je užitočné zaviesť hneď na začiatku jednotné názvoslovie. Tabuľka 3-1 obsahuje zoznam pojmov používaných ďalej s krátkymi popismi.

| **Pojem** | **Popis** |
|---|---|
| nástroj | výsledok tejto práce; Object Viewer |
| prostredie | vývojové prostredie Visual Studio alebo samostatná aplikácia; obe používajú nástroj Object Viewer |
| ladený proces | proces, ktorý je aktuálne ladený prostredím |
| vizualizér[2] | rozšírenie (plugin) samotného nástroja, pomocou ktorého je možné zobraziť obsah často používaných užívateľsky definovaných typov prívetivým spôsobom |
| procedúra | jednotné pomenovanie pre funkcie a metódy |
| blok | úsek kódu v procedúre ohraničený zloženými zátvorkami |
| typ | skalárny alebo užívateľsky definovaný typ |
| objekt | inštancia typu |
| premenná | pomenovaný objekt |
| plocha | grafický priestor obsahujúci objekty |
| rámec | reprezentuje objekt v nástroji; môže obsahovať vnorené rámce; na obrázku 3-2 obsahuje rámec pre objekt triedy `Dog` 4 rámce pre objekty `name`, `weight`, `color`, `owner`; rámec pre `color` obsahuje ďalšie 3 rámce pre objekty `red`, `green`, `blue` |
| skupina | rámec, ktorý sa nenachádza v nijakom inom rámci, ale priamo na ploche; je možné ho na rozdiel od jeho vnorených rámcov priamo presúvať na ploche |
| linka | reprezentuje ukazovateľ v nástroji (graficky vo forme orientovanej hrany) |
| kompozitný rámec | rámec, ktorý má viac vlastníkov (kapitola 2.4) |
| názov rámca | názov premennej alebo adresa objektu |
| typ rámca | názov typu objektu |
| hodnota rámca | úplná alebo náhľadová hodnota objektu |
| rozbalovací rámec | rámec, ktorý môže obsahovať vnorené rámce (napr. objekt triedy) |
| rozbalený rámec | rámec, ktorý má viditeľný zoznam vnorených rámcov |
| linkovateľný rámec | rámec, ktorý môže ukazovať na iný rámec (napr. ukazovateľ) |
| linkujúci rámec | rámec, ktorý aktuálne ukazuje na iný rámec |

**Tabuľka 3-1: Definícia pojmov**

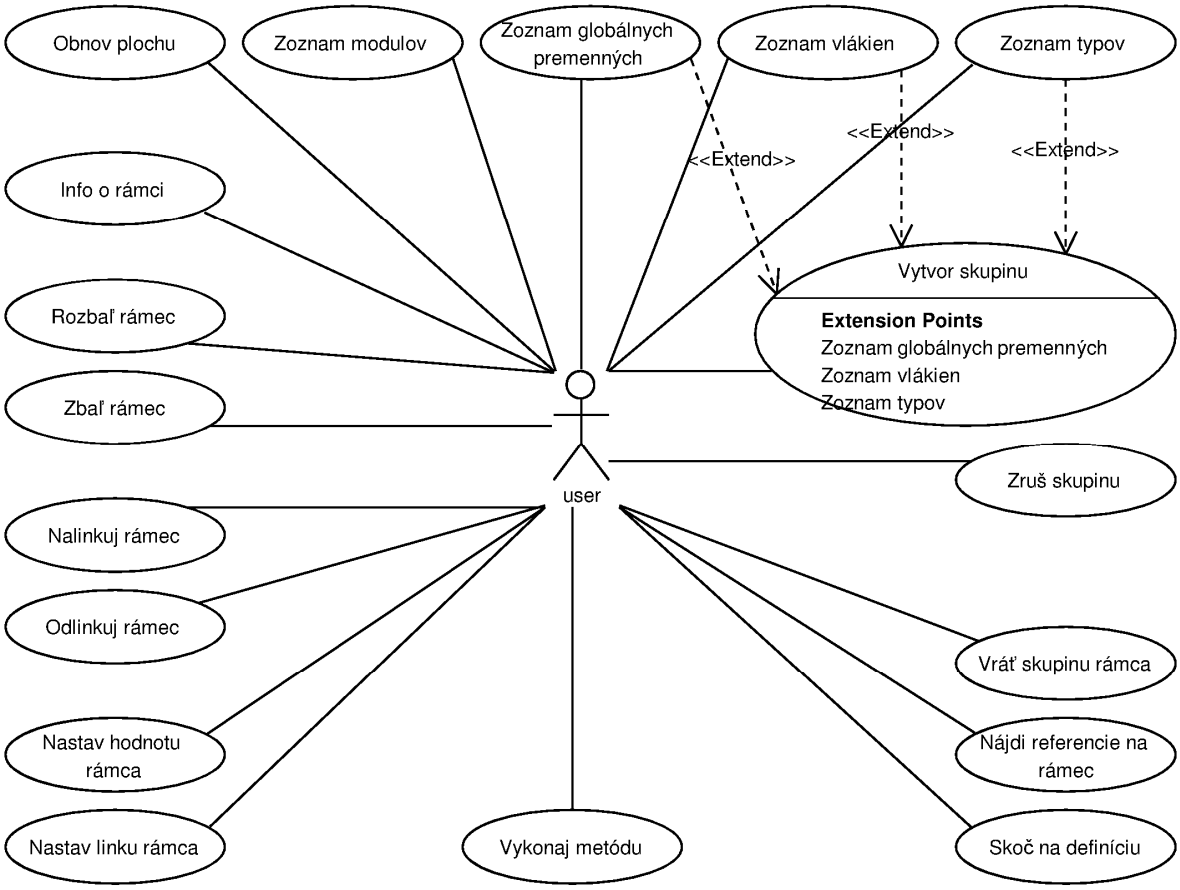
[2] Pojem vizualizér (angl. visualizer) je používaný na podobné účely v prostredí Visual Studio [14].

## 3.3 Diagram prípadov použitia

Obrázok 3-4 obsahuje diagram prípadov použitia nástroja. Keďže nemá zmysel zavádzať role užívateľov, obsahuje len jedného aktéra – user. Prípad použitia Obnov plochu v sebe zahrňuje každý prechod ladeného procesu do pozastaveného stavu. Informácie o rámci obsahujú jeho názov, adresu, typ, hodnotu, referenciu na iný rámec a zoznam vnorených rámcov.

Novú skupinu je možné vložiť na plochu podľa toho čo má reprezentovať: globálnu premennú, vlákno, alebo objekt ľubovoľného typu na ľubovoľnej adrese. Vhodný dialóg zobrazí všetky možnosti a užívateľ si vyberie skupiny, ktoré si želá pridať na plochu. Prípad použitia Vráť skupinu rámca zahrňuje pokus o nájdenie a zobrazenie najvyššieho rámca pre daný rámec – skupinu. Prípad použitia Nájdi referencie na rámec nájde a zobrazí všetky rámce (a ich linky) ukazujúce na daný rámec.

Ďalšie prípady použitia popisujú stav rozbalenia a linkovania (ukazovateľ na iný rámec) rámca a potrebu zmeniť hodnotu rámca a rámec, na ktorý ukazuje. Nástroj umožní vykonanie ľubovoľnej metódy nad vybraným objektom.




**Obrázok 3-4: Diagram prípadov použitia**

## 3.4 Analýza požiadaviek

Aplikácia bude v konečnej podobe rozšírením vývojového prostredia Visual Studio. Visual Studio poskytuje tri možnosti ako rozšíriť jeho funkčnosť. Je možné použiť *makrá* (macros), *rozšírenia* (add-ins) a *balíčky* (packages). Makrá sa používajú na jednoduché úlohy. Majú prístup k rozhraniu Code-Model. Nie je však možné pomocou nich vytvárať vlastné okná nástrojov (tool windows). Rozšírenia túto funkčnosť obsahujú. Balíčky sú v tomto smere najsilnejšie a umožňujú napríklad vytváranie vlastných typov dokumentov. Schopnosť vytvárať okná nástrojov je postačujúca pre tento projekt, preto bude pri integrácií do vývojového prostredia použité rozšírenie.

Pri návrhu aplikácie bude treba zohľadniť niektoré vlastnosti jazykov C a C++. Ide napríklad o typy `union`, pri ktorých sa viac objektov (potenciálne rovnakého typu) alebo členských premenných nachádza na rovnakých adresách. Tiež nie je možné za normálnych okolností rozlíšiť, či daný ukazovateľ ukazuje na jeden objekt, alebo na pole objektov a podobne nie je možné určiť veľkosť tohto poľa.

Aplikácia bude používať niekoľko rozhraní. Jedná sa o DIA a CodeModel. Bude potrebné pochopiť spôsob uloženia dát v nich. Z tohto dôvodu budú vytvorené krátke programy, ktoré dáta uchovávané v týchto rozhraniach zapíšu do jednoduchých textových súborov.

Keďže aplikácia zobrazuje graficky obsah pamäte ladeného procesu, musí mať prístup k jeho pamäti a k registrom jednotlivých vlákien. Počas analýzy vývojového prostredia Visual Studio sa nepodarilo zistiť spôsob ako rýchlo pristúpiť k pamäti ladeného procesu a k obsahom registrov. Jediný spôsob v tomto smere predstavujú rozhrania na vyhodnotenie výrazov [15], čo by ale neúmerne spomalilo beh nástroja, keďže sa výrazy kompilujú. Je ale možné použiť natívne funkcie Windows API, ako napríklad `ReadProcessMemory` a `GetThreadContext`. Rozhrania na vyhodnotenie výrazov sa použijú pri vykonávaní metód nad objektmi.

Nástroj zobrazuje graf objektov. Graf obsahuje prvky, ktoré v sebe obsahujú ďalšie prvky a texty. Medzi prvkami grafu sa nachádzajú orientované hrany. Aplikácia by mala obsahovať možnosť priblížiť a oddialiť pohľad. Túto funkčnosť je možné nájsť v niekoľkých grafických knižniciach. Nakoniec bola vybraná voľne dostupná knižnica Piccolo2D [16].

## 3.5 Naplánovanie projektu

Vývoj aplikácie bude prebiehať v niekoľkých etapách. Najprv sa vytvoria dva jednoduché programy na uloženie dát z rozhraní DIA a CodeModel do textových súborov. Tie budú mať názorný charakter. Výstup programov je potom možné použiť pri ďalšom vývoji pri hľadaní problémov ale aj pre získanie lepšej predstavy o vlastnostiach a možnostiach týchto dvoch rozhraní.

Pokračovať sa bude vývojom samotného *jadra* aplikácie, ktoré sa bude nachádzať v samostatnom programe. Jadro sa napojí na vybraný proces (typicky ladený v prostredí Visual Stu-

dio). Na žiadosť užívateľa graficky zobrazí alebo obnoví obsah jeho pamäte na ploche. Nezávislosť jadra na vývojovom prostredí uľahčí vývoj a v budúcnosti umožní použitie nástroja aj v iných prostrediach.

Následne sa vytvorí rozšírenie prostredia Visual Studio, ktoré bude používať prostriedky jadra. Bude vytvorené vlastné okno nástrojov obsahujúce prvky jadra. Obnovenie plochy sa udeje vždy pri prechode do pozastaveného stavu.

Potom sa vytvoria vizualizéry pre typy z knižnice STL. Ide hlavne o kontajnery a iterátory. Vizualizéry budú existovať vo forme rozšírenia samotného nástroja v samostatnej knižnici.

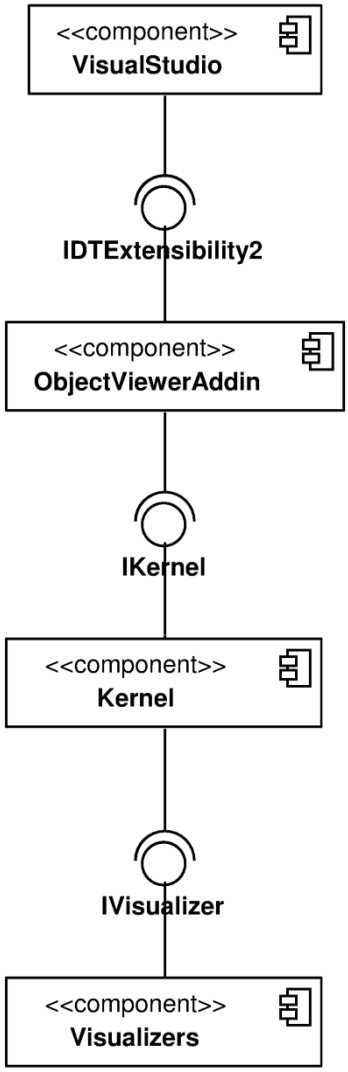
Nakoniec sa pridajú funkcie na nájdenie skupín a všetkých referencií. Aplikácia sa ešte viac integruje pridaním funkcií na prechod k definíciám tried, premenných a metód a pridaním možnosti vykonávať metódy. Prebehne grafická úprava a spríjemnenie užívateľského rozhrania.

## 3.6 Diagram komponentov

Diagram komponentov celého systému je znázornený na obrázku 3-5. Obrázok znázorňuje vzťah medzi existujúcimi komponentmi a komponentmi vytvorenými v tejto práci. Jadro nástroja sa nachádza v komponente Kernel, ktorý je reprezentovaný rozhraním `IKernel`. Ten je možné použiť samostatne, alebo ako súčasť vývojového prostredia pomocou komponentu ObjectViewerAddin. Rozšírenie prostredia Visual Studio implementuje rozhranie `IDTExtensibility2`. Komponent VisualStudio zastupuje prostredie Visual Studio.

Rozšírenia samotného nástroja sú reprezentované komponentom Visualizers a implementujú rozhranie `IVisualizer`.

Najdôležitejším a zároveň najkomplexnejším komponentom systému je jadro Kernel, preto bude zvyšná časť návrhu venovaná prevažne jemu.

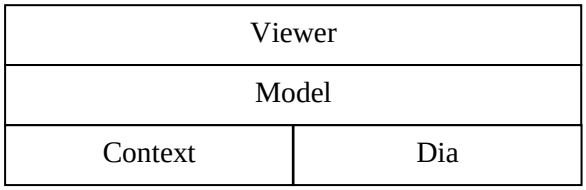



**Obrázok 3-5: Diagram komponentov**

## 3.7 Návrh architektúry

Jadro obsahuje vnútornú reprezentáciu kontextu ladeného procesu a tiež prostriedky na jej grafické zobrazenie. Je teda hneď možné rozlíšiť dve vrstvy jadra: model a pohľad. Model zapuzdruje kontext ladeného procesu spolu so symbolickými informáciami.

Tieto skutočnosti sú znázornené na obrázku 3-6. Obrázok znázorňuje jednotlivé vrstvy jadra. Jadro sa skladá z troch vrstiev. Najvyššia vrstva (Viewer) sa stará o grafické zobrazenie dát z vrstvy modelu (Model). Kontext ladeného programu (Context) a symbolické informácie (Dia) sú dve rovnocenné sekcie na najnižšej vrstve. Jedna je totiž zdrojom dát, druhá zdrojom ich sémantiky.



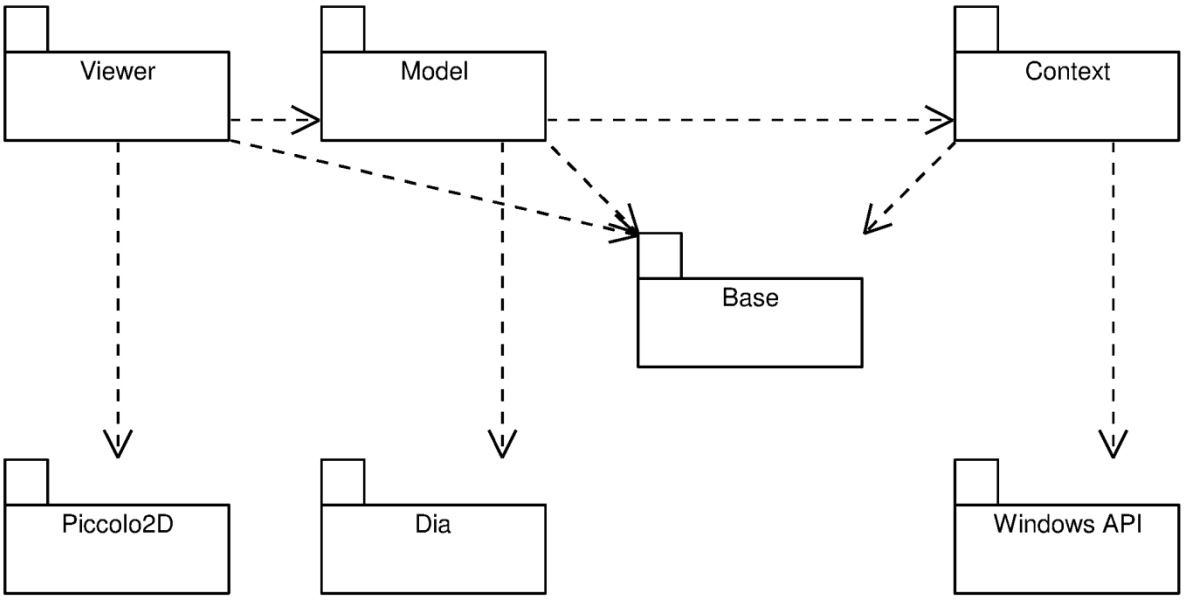

**Obrázok 3-6: Architektúra jadra**

V praxi môže byť počet objektov v pamäti vysoký. Pohľad však bude obsahovať väčšinou len niekoľko objektov dôležitých v aktuálnej situácií. Preto je vhodné aby model obsahoval vnútornú reprezentáciu len tých objektov, ktoré sú viditeľné v pohľade. Taktiež nie je nutné aby model obsahoval kópiu dát z kontextu, pretože tie môže kedykoľvek získať explicitne. V konečnom dôsledku bude teda model obsahovať dvojice [adresa, symbol]. Táto dvojica dostatočne popisuje objekt. Dáta objektu môže model kedykoľvek získať z nižších vrstiev.

Väčšinou bude v systéme existovať len jedna inštancia modelu (aj keď to nie je nutná podmienka). Pohľadov však môže existovať niekoľko. Núka sa možnosť použitia architektonického vzoru MVC (Model-view-controller), kde pohľad a kontrolór budú obsiahnuté vo vrstve Viewer a model vo vrstve Model. Dajú sa tu aplikovať všetky princípy tohto vzoru. Ak napríklad užívateľ zmení hodnotu objektu v pohľade, táto zmena je aplikovaná na model, ktorý ju aplikuje na dátovú vrstvu – kontext. Následne notifikuje všetky pohľady (napríklad pomocou návrhového vzoru Observer) a tie obnovia svoj obsah z modelu.

## 3.8 Diagram balíkov

Na obrázku 3-7 sa nachádza diagram balíkov jadra spolu so závislosťami medzi jednotlivými balíkmi. Balík Model obsahuje triedy, ktoré vytvárajú abstrakciu pamäte ladeného programu v podobe orientovaného grafu objektov. Je závislý primárne na dvoch balíkoch. Prvým je balík Dia, ktorý reprezentuje rozhranie Debug Interface Access. Druhým je balík Context, ktorý zapuzdruje kontext ladeného procesu. V diagrame je zobrazená symbolická závislosť balíka Context na balíku Windows API, keďže balík Context používa niekoľko natívnych funkcií. Balík Viewer zabezpečuje grafickú podobu balíku Model a preto na ňom závisí. Okrem toho závisí na externej grafickej knižnici Piccolo2D. Jednotlivé balíky závisia na balíku Base, ktorý definuje niektoré základné typy, používané v celom systéme.




**Obrázok 3-7: Diagram balíkov**

# 3.9 Diagramy tried

Táto kapitola obsahuje diagramy tried jadra. Kvôli väčšej prehľadnosti bol celkový diagram tried rozdelený na niekoľko menších. Diagramy sú usporiadané systémom „zdola nahor“, to zn. prvé sú popísané tie, ktoré nezávisia na žiadnych iných.

## 3.9.1 Balíky Base a Context

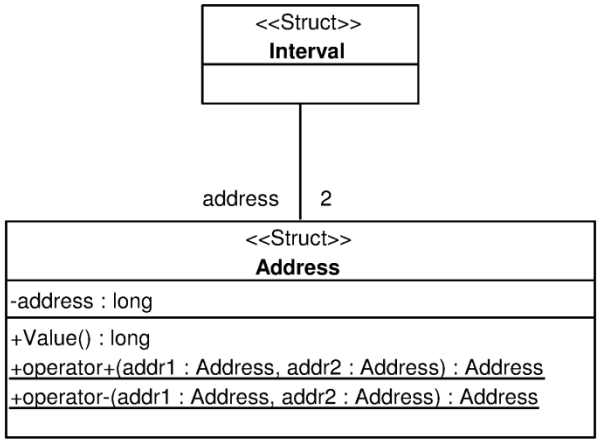
Na obrázku 3-8 sa nachádza diagram tried pre balík Base. Tento balík obsahuje 2 triedy používané v celom systéme. Trieda `Address` zapuzdruje virtuálnu adresu, veľkosť alebo akúkoľvek inú skalárnu hodnotu. Trieda `Interval` zapuzdruje súvislý rozsah (interval) virtuálnych adries. Trieda `Address` v konečnom dôsledku obsahuje atribút jednoduchého dátového typu. Je navrhnutá preto, aby bolo možné zmeniť v budúcnosti tento dátový typ (napríklad pri prechode na architektúru procesora s inou šírkou slova).

Trieda `Address` obsahuje sadu štandardných operátorov. Ide o najmä o súčet a rozdiel. Kvôli jednoduchosti na obrázku chýbajú ďalšie preťažené operátory, napríklad operátor porovnania.

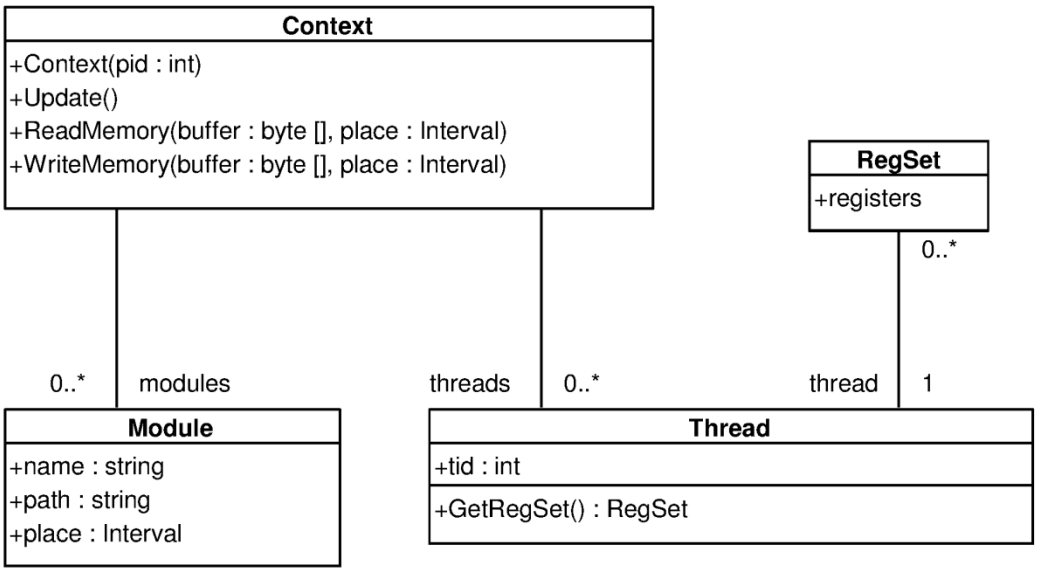
Bolo by možné rozdeliť funkčnosť triedy `Address` do dvoch tried. Tou druhou by bola trieda `Size`. Odčítaním dvoch adries by vznikla práve inštancia tejto triedy. V konečnom dôsledku by to ale pravdepodobne viedlo na neustále pretypovanie z jedného typu na druhý.

Na obrázku 3-9 sa nachádza diagram tried pre balík Context. Triedy v tomto balíku zapuzdrujú kontext ladeného procesu. Vstupnou triedou je trieda `Context`. Obsahuje operácie na prístup do pamäti procesu (`ReadMemory`, `WriteMemory`). Ďalej obsahuje aktuálnu sadu modulov (`modules`) a vlákien (`threads`). Trieda `Module` zastupujúca modul obsahuje atribúty názvu súbo-

ru, jeho cesty a umiestnenia v pamäti. Trieda `Thread` zastupujúca vlákno obsahuje atribút identifikátora vlákna. Jej operácia `GetRegSet` vracia aktuálnu sadu registrov zastúpenú triedou `RegSet`. Každá funkcia na zásobníku vlákna má jedinečnú sadu registrov.




**Obrázok 3-8: Diagram tried pre balík Base**




**Obrázok 3-9: Diagram tried pre balík Context**

Trieda `Context` obsahuje operáciu `Update`, ktorá obnoví uschované dáta: sadu modulov a vlákien. Obsahuje konštruktor, ktorý vytvorí jej objekt na základe identifikátora procesu. V budúcnosti môže obsahovať aj konštruktor, ktorý akceptuje iný zdroj dát (napríklad súbory minidump [17]) alebo je možné návrh upraviť vytvorením konkrétnych tried pre jednotlivé zdroje dát.

### 3.9.2 Balík Model

Na obrázku 3-10 sa nachádza diagram tried pre balík Model. Diagram obsahuje aj niektoré triedy z balíkov Context a Dia. Podobne ako v predchádzajúcich diagramoch sú aj v tomto kvôli väčšej prehľadnosti vynechané detaily (menej dôležité triedy, vzťahy, atribúty a operácie).

Trieda `OID` slúži na jednoznačnú identifikáciu objektov ladeného procesu. Okrem samotných objektov však jednoznačne identifikuje aj vlákna a procedúry na zásobníku. Skladá sa z dvoch prvkov: adresy (`address`) a symbolu (`symbol`). Ako bolo ukázané v kapitole 2.4 táto dvojica jednoznačne identifikuje konkrétny objekt v pamäti. Operácie `Equals` a `GetHashCode` definujú totožnosť objektov triedy `OID`.

Typy objektov sú reprezentované inštanciami triedy `Symbol`. Trieda `Symbol` rozširuje rozhranie `IDiaSymbol` o ďalšie atribúty. V rozhraní Dia môžu existovať duplicitné symboly (typicky pri symboloch užívateľsky definovaných typov). Tieto však majú priradenú jedinú inštanciu triedy `Symbol` pretože ide v konečnom dôsledku o ten istý dátový typ.

Modul (`Module`) a k nemu prislúchajúci zdrojov symbolov (`IDiaDataSource`) sú spojené v triede `SymbolSource`. Jeden modul má potom niekoľko symbolov (trieda `Symbol`). Jadro obsahuje len tie symboly, pre ktoré existuje aspoň jedna inštancia triedy `OID`.

Implementácie rozhrania `IVisualizer` (vizualizéry) poskytujú prístup k vlastnostiam objektov. Každý symbol má priradený práve jeden vizualizér. Symbolov je v normálnom prípade oveľa viac ako vizualizérov. Napríklad symboly pre typy `std::list<int>` a `std::list<float>` môžu mať priradený jeden spoločný vizualizér. Typickou operáciou vizualizéra je `GetValueString`, ktorá vráti textovú reprezentáciu hodnoty konkrétneho objektu v pamäti. Jadro obsahuje jeden spoločný, implicitný vizualizér. Užívateľ si ale môže pripojiť vlastné vizualizéry pre svoje typy, ktoré používa.

Operácia `GetValueString` je len jednou z operácií, ktoré sú definované v rozhraní `IVisualizer`. Medzi podobné operácie patria `GetNameString` a `GetTypeString`, ktoré vracajú textovú reprezentáciu názvu premennej a typu objektu. Operácia `GetChilds` vracia kolekciu vnorených objektov a operácia `GetLink` vracia objekt, na ktorý daný objekt ukazuje. Pomocou operácií `SetValueString` a `SetLink` môže užívateľ meniť hodnotu objektu.

Zatiaľ čo trieda `OID` reprezentuje nižšie vrstvy modelu, trieda rámca `Frame` vytvára most medzi `OID` a pohľadom. Všetky pohľady (užívateľ môže mať otvorených niekoľko pohľadov) a všetky grafické objekty v pohľadoch sú zastúpené rámcami. Každý rámec reprezentuje práve jeden objekt pomocou inštancie `OID`. Rámce môžu byť do seba vnorené a ukazovať na seba.

Atribúty `expanded` a `linked` obsahujú aktuálny stav rozbalenia a linkovania rámca. Trieda `Frame` ďalej obsahuje kópiu operácií z rozhrania `IVisualizer` (v diagrame sú kvôli prehľadnosti vynechané). Jediným rozdielom je, že neobsahujú parameter typu `OID`, keďže ten je zakomponovaný

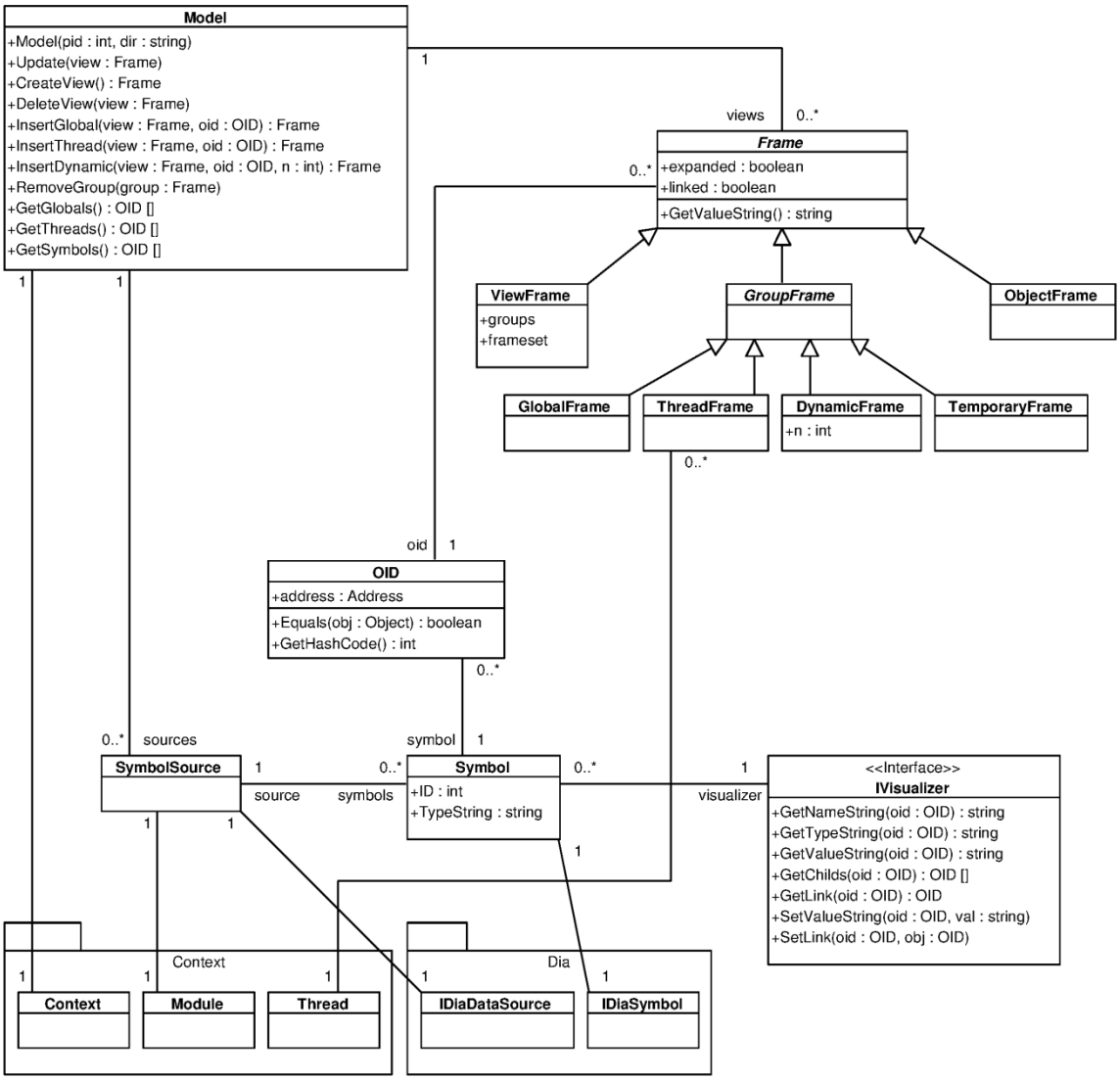
v samotnej triede `Frame`. Tieto operácie vedú na volanie príslušných operácií z priradeného vizualizéra `IVisualizer`.

Trieda `Frame` je abstraktnou bázovou triedou. Zdedená trieda `ViewFrame` zastupuje pohľady. Každý pohľad obsahuje ako koreňový rámec inštanciu tejto triedy. Koreňový rámec ďalej obsahuje niekoľko skupín (trieda `GroupFrame`), ktoré si na plochu pridáva užívateľ: `GlobalFrame` pre globálne premenné, `ThreadFrame` pre vlákna a `DynamicFrame` pre dynamické objekty. Trieda `TemporaryFrame` reprezentuje dočasnú skupinu. Tá sa automaticky vytvorí napríklad vtedy, ak užívateľ nechá zobraziť cieľ linky, ktorý ale nemá explicitne pridaný na ploche ako skupinu. Všetky premenné a objekty sú reprezentované triedou `ObjectFrame`.

Atribút `n` v triede `DynamicFrame` obsahuje veľkosť poľa. Môže ho zadať užívateľ. Predpokladá sa, že má znalosť o tom, či sa na danej adrese nachádza len jeden objekt, alebo niekoľko objektov s rovnakým typom za sebou.

Trieda pohľadu `ViewFrame` obsahuje 2 atribúty. Atribút `groups` obsahuje kolekciu skupín (trieda `GroupFrame`). Atribút `frameset` obsahuje všetky inštancie triedy `ObjectFrame` v danom pohľade. Aj keď tvoria rámce hierarchickú štruktúru tento vzťah nie je explicitne uchovávaný v modeli. Podobne ako je to pri hodnotách objektov nie je potrebné túto informáciu uchovávať v tejto vrstve.

Vrstva modelu je zastúpená triedou `Model`. Pokyn na obnovu pohľadu vybavuje jej operácia `Update`. Užívateľ môže vytvárať prázdne pohľady (`CreateView`) a rušiť ich (`DeleteView`). Skupiny je možné pridávať (`InsertGlobal`, `InsertThread`, `InsertDynamic`) a existujúce skupiny rušiť (`RemoveGroup`). Kolekciu možných skupín vracajú operácie `GetGlobals`, `GetThreads` a `GetSymbols`.




**Obrázok 3-10: Diagram tried pre balík Model**

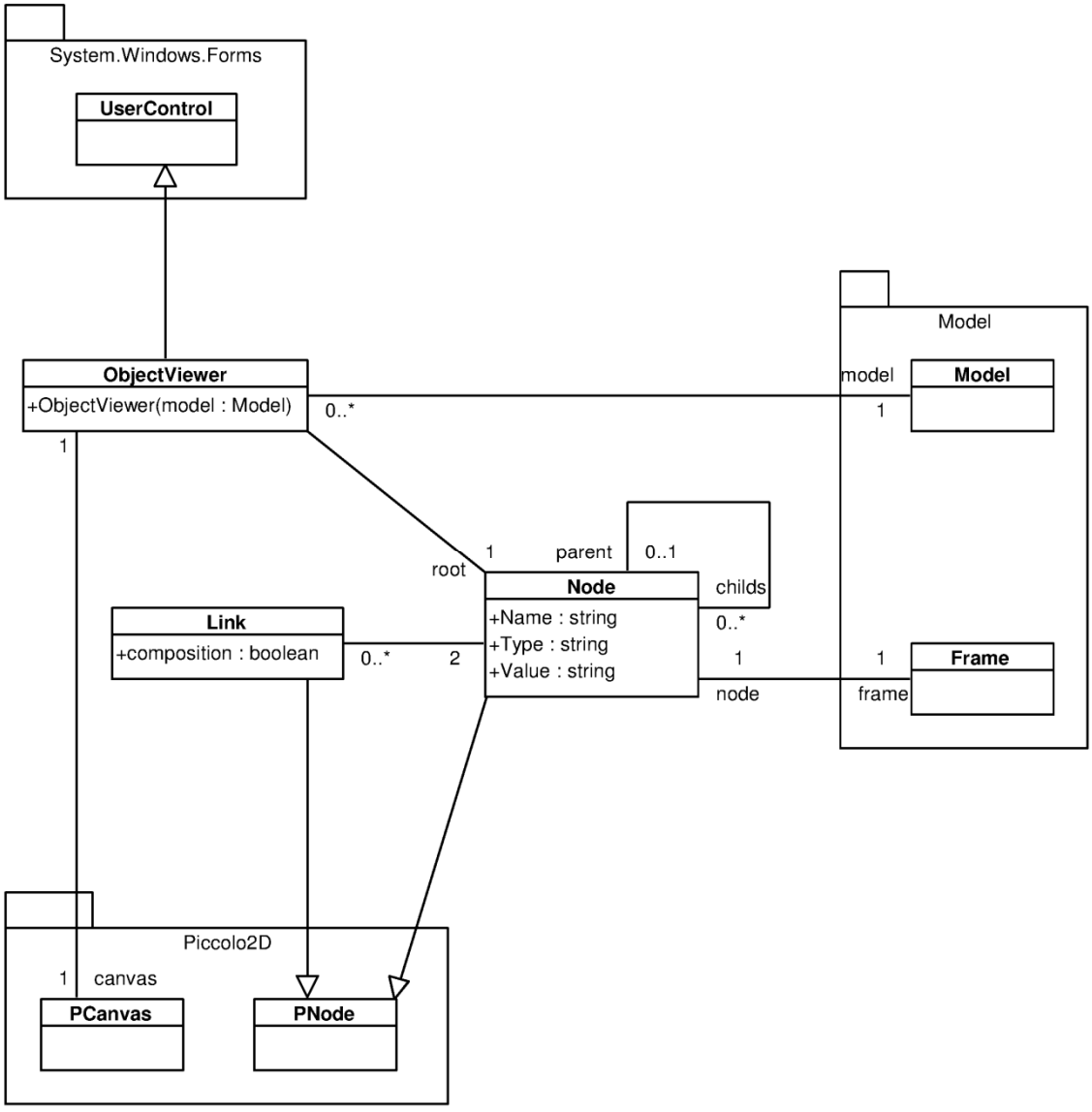
## 3.9.3 Balík Viewer

Balík Viewer pridáva k balíku Model vizuálnu časť. Diagram tried pre balík Viewer sa nachádza na obrázku 3-11. Hlavnou triedou je trieda `ObjectViewer`. Je to špecializovaná trieda triedy `UserControl`, takže ide o užívateľský prvok zobraziteľný v okne. Každá inštancia triedy `ObjectViewer` reprezentuje jeden pohľad. Obsahuje plátno (`PCanvas`), na ktorom sú zobrazené jednotlivé rámce. V určitom okamihu existuje v systéme jedna inštancia modelu a nad ňou niekoľko inštancií triedy `ObjectViewer`.

Trieda `Node` pridáva vizuálnu funkcionalitu triede `Frame` z modelu. Jednotlivé inštancie triedy `Node` tvoria strom (reflexívny vzťah `parent – childs`) s koreňovým uzlom ekvivalentným

s inštanciou triedy `ViewFrame`. Hlavnými atribútmi triedy `Node` sú názov, typ a hodnota objektu. Tie sú priamo prezentované užívateľovi.

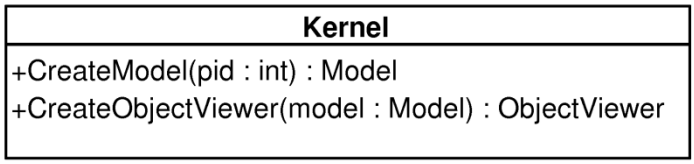
Medzi inštanciami triedy `Node` môžu existovať hrany, ktoré sú reprezentované inštanciami triedy `Link`. Jej atribút `composition` je použitý v špeciálnych prípadoch kedy niekoľko premenných obsahuje ten istý objekt. V takom prípade sa rámec pre objekt oddelí od rámcov pre premenné a vytvorí sa medzi nimi špeciálny vzťah kompozície vyjadrený hranou s atribútom `composition` rovným `true`.




**Obrázok 3-11: Diagram tried pre balík Viewer**

### 3.9.4 Rozhranie jadra

Z vonkajšieho pohľadu sa jadro skladá z modelu a pohľadu, ktoré sú reprezentované triedami `Model` a `ObjectViewer`. Na rozhranie jadra bol preto aplikovaný návrhový vzor Factory Method. Výsledok sa nachádza na obrázku 3-12.




**Obrázok 3-12: Rozhranie jadra**

## 3.10 Obnovenie rámcov

Pri prechode ladeného programu do pozastaveného stavu je vždy nutné obnoviť údaje v modeli a následne v pohľade. To sa udeje napríklad ak užívateľ krokuje program. Aj jednoduché preskočenie (angl. *step over*) riadku programu môže vyvolať veľké zmeny v pamäti programu. Zmenia sa hodnoty objektov, zmení sa počet prvkov v kontajneroch, prvky v kontajneroch sa presunú do iných kontajnerov, zmenia sa ukazovatele tak, že ukazujú na nové objekty, alebo niektoré objekty úplne zaniknú. Tak či onak sa celá hierarchická štruktúra rámcov zmení.

Jednou možnosťou ako tento problém vyriešiť by bolo zahodiť existujúce rámce a vytvoriť zakaždým nové. Takéto nové rámce by ale stratili užívateľom nastavené vlastnosti starých rámcov, napríklad polohu na ploche, stav rozbalenia a linkovania.

Lepšia možnosť je popísaná ďalej. Namiesto zahadzovania rámcov, sa tie zbavia svojej hierarchickej štruktúry a rovnocenne sa umiestnia dočasne na jednu *hromadu*. Následne jednoduchý grafový algoritmus prejde všetky skupiny a postupne opäť vytvorí štruktúru rámcov. Ak tento algoritmus narazí na objekt s nejakým `OID` pozrie sa na hromadu, či sa tam nachádza rámec s rovnakým `OID`. Ak áno vyberie ho z hromady a umiestni ho do novej štruktúry rámcov. Ak nie, vytvorí nový rámec. Na konci zostanú v hromade rámce, ktoré neboli nijak odkazované a môžu byť odstránené. Tento jednoduchý postup zabezpečí to, že všetky rámce, ktoré „prežili“ chod programu zostanú aj v novej štruktúre.

Nakoniec je vhodné povedať, že hromada, ktorá tu bola spomenutá je vlastne atribút `frameset` v triede `VievFrame` v balíku Model. Táto trieda obsahuje aj zoznam skupín v atribúte `groups`.

# 4 Implementácia

Táto kapitola popisuje implementačnú časť tejto práce. Sú tu popísané jednotlivé fázy vývoja tak ako boli navrhnuté v kapitole 3.5. Nezachádza príliš do detailov, pretože to je úlohou iných dokumentov, napríklad programovej dokumentácie (príloha A). Všetky časti práce boli implementované vo vývojovom prostredí Visual Studio v jazykoch C# a C++.

## 4.1 Utilita diadump

Ešte pred samotným nástrojom boli implementované dva jednoduché programy. Prvým je program diadump, ktorý exportuje symbolické informácie patriace nejakému spustiteľnému súboru do textovej podoby. Program sa spúšťa z príkazového riadku jedným z nasledujúcich spôsobov:

```
diadump stat <súbor>
diadump dyn <súbor>
```

Prvý spôsob exportuje symbolické informácie spustiteľného súboru (typicky exe alebo dll) alebo súboru zo symbolickými informáciami (typicky pdb) do textového súboru dump.txt a tento súbor následne otvorí. Každý riadok súboru obsahuje dáta o jednom symbole. Hierarchická štruktúra symbolov je uložená pomocou odsadenia. Na obrázku 4-1 je zobrazený krátky výpis z tohto súboru.

```
@15 SYM_FUNCTION [name=main] [rva=00011370] [length=0000002c]
    @19 SYM_DATA [name=agrc] [type=20]
        [location=regrel][rid=EBP][off=00000008]
    @21 SYM_DATA [name=argv] [type=22]
        [location=regrel][rid=EBP][off=0000000c]
    @23 SYM_DATA [name=fit] [type=24]
        [location=regrel][rid=EBP][off=fffffff4]
@20 SYM_BASETYPE [base-type=int] [length=00000004]
@22 SYM_POINTERTYPE [target-type=6356] [length=00000004]
@6356 SYM_POINTERTYPE [target-type=6448] [length=00000004]
@6448 SYM_BASETYPE [base-type=char] [length=00000001]
@24 SYM_BASETYPE [base-type=float] [length=00000008]
```

**Obrázok 4-1: Statické symbolické informácie**

Každý riadok začína identifikačným číslom symbolu a jeho typom. Nasleduje niekoľko atribútov, kde každý atribút je tvorený názvom a hodnotou. Na obrázku sú pre názornosť zámerne vybrané riadky a ich atribúty, ktoré súvisia s funkciou main. V skutočnosti má takýto textový súbor veľkosť niekoľkých megabajtov.

Druhý spôsob spustenia zachytáva dynamické informácie. Pri tomto spôsobe diadump spustí daný súbor v ladiacom režime. Zakaždým keď sa program pokúsi vykonať inštrukciu s kódom 0xCC[3] diadump zapíše do súboru obsah zásobníka. Krátky výpis z výsledného súboru je zobrazený na obrázku 4-2. V danej chvíli sú na zásobníku aktívne dve knižničné funkcie a funkcia main.

```
[func_ip=00aa1828] [func-name=main][tag=SYM_FUNCTION]
[func_ip=00aa1718] [func-name=__tmainCRTStartup][tag=SYM_BLOCK]
[func_ip=00aa155f] [func-name=mainCRTStartup][tag=SYM_FUNCTION]
```

**Obrázok 4-2: Dynamické symbolické informácie**

Súčasťou programu diadump je aj ukážková aplikácia a dva súbory dump_stat.cmd a dump_dyn.cmd, ktoré spustia program diadump s pripravenými argumentmi.

Program je implementovaný v jazyku C++. Výsledný binárny súbor (diadump.exe) potrebuje na svoju činnosť samotnú knižnicu DIA (msdia90.dll), ktorú ale nie je nutné explicitne registrovať v systéme.

## 4.2 Utilita modeldump

Druhým prípravným programom je program modeldump. Exportuje zdrojové súbory do textového súboru vo forme abstraktného syntaktického stromu a ten následne otvorí. Program je implementovaný ako rozšírenie prostredia Visual Studio. Vstupom programu je aktuálny projekt otvorený v prostredí. Aktivuje sa pomocou položky Model Dumper z menu nástrojov (Tools).

Formát súboru je podobný ako pri programe diadump. Jedným z rozdielov je fakt, že element môže obsahovať viac zoznamov elementov. Element pre triedu má napríklad zoznam členských premenných, zoznam šablónových parametrov a zoznam bázových tried. Na obrázku 4-3 je zobrazený krátky výpis z tohto súboru. Pre názornosť sú zobrazené len niektoré elementy a atribúty.

```
[kind=vsCMElementFunction][name=main]
    parameters:
        [kind=vsCMElementParameter][name=agrc]
        [kind=vsCMElementParameter][name=argv]
[kind=vsCMElementFunction][name=create]
[kind=vsCMElementFunction][name=go]
```

**Obrázok 4-3: Výstup programu modeldump**

Program taktiež vytvorí záložku CodeModel Events v okne Output, kde sa zobrazujú udalosti generované pri zmene modelu. Na obrázku 4-4 sa nachádza časť z výpisu.

[3] Ladiace programy ju používajú na prerušenie behu (angl. breakpoint)

```
Added: vut, vsCMElementFunction
Changed: vut2000, vsCMElementFunction, vsCMChangeKindRename
Deleted: vut2000, vsCMElementFunction, System.__ComObject
```

**Obrázok 4-4: Udalosti kódového modelu**

Program je nutné zaregistrovať v prostredí Visual Studio. Všetko čo je potrebné urobiť je skopírovať súbory modeldump.dll a modeldump.AddIn do adresára %documents%\Visual Studio 2008\Addins, kde %documents% je adresár dokumentov jedného alebo všetkých užívateľov. Pri použití prostredia Visual Studio inej verzie ako 2008 je potrebné v ceste zmeniť 2008 na danú verziu. Súčasťou sú aj zdrojové kódy v jazyku C#.

## 4.3 Prototyp

Prvou úlohou bolo rýchle vytvorenie prototypu nástroja v implementačnom jazyku C#. Z celého návrhu bolo najprv implementované jadro, respektíve jeho najnutnejšia časť. Z každej vrstvy jadra sa implementovala najnutnejšia časť tak aby výsledný program zobrazoval aspoň niečo.

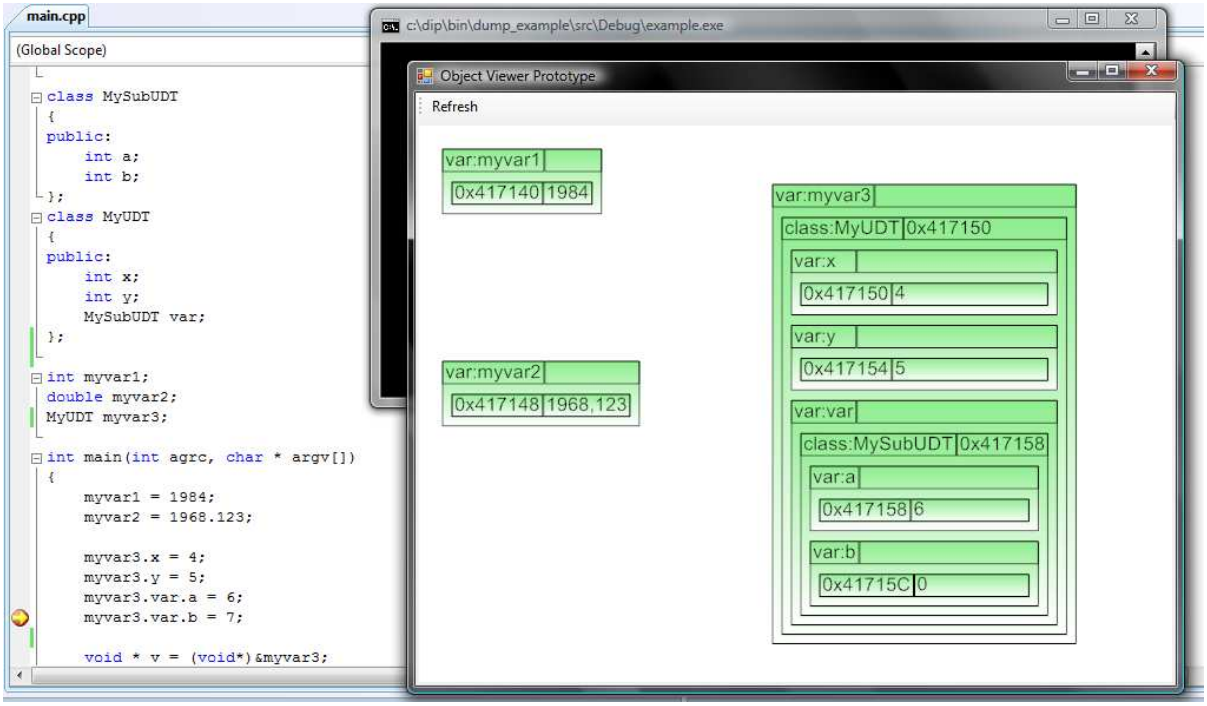
Úlohou prototypu bolo zobrazenie niekoľkých dopredu zadaných globálnych premenných v grafickej podobe. Výsledné objekty boli vždy rozbalené, bez možnosti ich zbaliť. Zatiaľ boli podporované len jednoduché dátové typy a niektoré skalárne typy, t.j. žiadne ukazovatele. Prototyp otvoril existujúci proces s názvom „example“ a načítal symbolické informácie pre základný modul (súbor exe).

Ukážka programu sa nachádza na obrázku 4-5. Tlačidlo Refresh obnoví hodnoty v diagrame podľa aktuálneho stavu ladeného programu (v pozadí obrázku). Na obrázku je vidieť niekoľko rámcov, ktoré zastupujú premenné a objekty. Platí, že objekt v sebe neobsahuje priamo vnorené objekty, ale je medzi nimi vždy ešte rámec pre premennú. Neskôr bude ukázané, že toto striedanie objektov a premenných je nevyhnutné a používa sa v celom systéme. Čísla v hexadecimálnej sústave sú adresy objektov v pamäti.

Rozhranie DIA existuje vo forme COM rozhraní. Na to aby ho bolo možné použiť v implementačnom jazyku C# ho bolo nutné skonvertovať do binárnej podoby pre toto prostredie (assembly). To je možné urobiť v dvoch krokoch pomocou programov midl a tlbimp, ktoré sú dodávané s prostredím Visual Studio. Týmto sa ale stratí časť informácie a tak má výsledný modul niekoľko nedostatkov. Neobsahuje všetky rozhrania a konštanty a niektoré parametre metód majú nesprávny typ. Ide hlavne o parametre s typom poľa, ktoré sa po skonvertovaní stanú obyčajnými referenciami na jeden prvok poľa. Tieto nedostatky je síce možné dodatočne vyriešiť ale nakoniec bol použitý čistejší spôsob a všetky rozhrania a konštanty boli explicitne nadefinované v implementačnom jazyku.

V každom prípade ide len o COM rozhrania a samotný kód rozhrania DIA sa nachádza v súbore msdia90.dll. Za normálnych okolností je nutné ho zaregistrovať v systéme. Bol ale použitý spôsob, ktorý registráciu nevyžaduje. Inšpiráciou bol natívny kód funkcie `NoRegCoCreate` z rozhrania DIA.

V tomto okamihu vývoja bola vytvorená kostra programu. V ďalších fázach boli do tejto kostry postupne doplňané ďalšie časti.



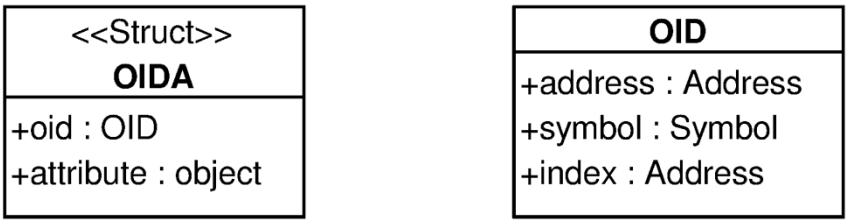

**Obrázok 4-5: Prototyp**

# 4.4 Triedy OID a OIDA

Trieda `OID` slúži na jednoznačnú identifikáciu objektu. Bola navrhnutá ako dvojica [adresa, symbol]. Vo väčšine prípadov je to postačujúce. Sú však situácie (typicky pri poliach), kedy len táto dvojica nestačí. Preto bola jednoznačná identifikácia `OID` rozšírená o tretí rozmer, o index. V konečnom dôsledku je to teda trojica [adresa, symbol, index]. Obrázok 4-6 ukazuje upravenú triedu `OID`.

Index sa používa pri poliach, pri premenných TLS a môžu ho slobodne používať externé vizualizéry. Na obrázku 4-7 sa nachádza jednoduchý typ reprezentujúci farbu. Za normálnych okolností by sa zobrazil tak ako na obrázku 4-8. Ak by však bola požiadavka zobraziť jeho farebné komponenty napríklad tak ako na obrázku 4-9 museli by sa vytvoriť fiktívne objekty pre jednotlivé farebné komponenty. Všetky takéto objekty sa ale musia líšiť aspoň v jednej zložke `OID`. Zmenou adresy by mohla nastať kolízia s inými objektmi. Rozdielne symboly by v tomto prípade pomohli, ale problém by vznikol pri poliach, kedy by bolo potrebné vygenerovať jedinečný symbol pre každý index každého

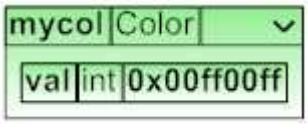
poľa. Preto bol tento problém vyriešený práve pridaním indexu do `OID`. V tomto príklade by mal základný objekt implicitný index 0 a jeho farebné komponenty napríklad 1, 2, 3.




**Obrázok 4-6: Triedy OIDA a OID**

```
struct Color
{
    int val;
};
```

**Obrázok 4-7: Typ Color**




**Obrázok 4-8: Objektu typu Color**


**Obrázok 4-9: Farebné komponenty objektu typu Color**

Pridanie indexu nerieši ešte jednu situáciu. Občas je totiž potrebné preniesť komplexnejšiu informáciu z predka na potomka, ale zároveň táto informácia logicky nijak neprispieva k jednoznačnej identifikácii objektu. Typickým príkladom je vlákno a jeho zásobník. Vlákno obsahuje aktívne procedúry na zásobníku vo forme potomkov a tie následne obsahujú lokálne premenné. Prvotnú sadu hodnôt registrov má vlákno. Vlákno musí jej hodnoty vhodne upraviť pre každú aktívnu procedúru na zásobníku a vložiť jej ju „do vienka“. Na druhej strane procedúry očakávajú túto sadu a podľa nej

určia adresy lokálnych premenných na zásobníku. Trieda `OIDA` (obrázok 4-6) zapuzdruje triedu `OID` spolu s ďalším voliteľným atribútom. Na rozdiel od indexu atribút neprispieva do jednoznačnej identifikácie objektu, preto ani nie je súčasťou samotnej triedy `OID`.

## 4.5 Triedy Symbol a SymbolSource

Triedy `Symbol` a `SymbolSource` zapuzdrujú rozhranie DIA. Zároveň `Model` obsahuje v sebe zoznam inštancií týchto tried. Toto sa ukázalo ako veľmi dobré riešenie. Rozhranie DIA je totiž kolekciou COM rozhraní. To znamená, že sa tam používa počítanie referencií (*reference counting*). Keď DIA otvorí konkrétny súbor so symbolickými informáciami, nie je ho možné zatvoriť inak než zrušením všetkých referencií na prislúchajúce rozhrania z DIA. Jazyk C# a prostredie .NET však používajú automatickú správu pamäte označovanú anglickým výrazom *Garbage Collection*. Ak by sa v rôznych častiach nástroja namiesto referencií na `Symbol` používali priamo referencie do rozhrania DIA, nebolo by možné zaručiť zrušenie týchto referencií a následné zatvorenie súboru so symbolickými informáciami. Výsledkom by bolo, že po ukončení ladenia by kompilátor nevedel otvoriť súbor so symbolickými informáciami na zápis a opätovný preklad by sa nepodaril. Tým, že `Model` obsahuje zoznam všetkých inštancií triedy `Symbol`, môže jednoduchým priechodom zrušiť všetky referencie do rozhrania DIA. Je možné, že nejaké časti nástroja stále držia referencie na inštancie triedy `Symbol`, tie už ale nedržia referencie do DIA.

Jazyky C a C++ rozlišujú medzi konštantnými a nekonštantnými typmi. To znamená, že z pohľadu rozhrania DIA sú typy `int` a `const int` rôzne. Ak by sa táto vlastnosť presunula aj do nástroja, mohlo by sa napríklad stať, že by na ploche boli zobrazené dva rôzne objekty s rovnakou adresou, pričom jeden by mal typ `int` a druhý `const int`. Toto by sa stávalo dosť často. Bežne sa totiž do funkcie očakávajúcej ukazovateľ na konštantný typ predáva ukazovateľ na nekonštantný typ.

Rozhranie DIA zároveň obsahuje duplikáty symbolov, ktoré sú logicky totožné. Ak je na jednom mieste programu použitý ukazovateľ na typ `int` a taký istý ukazovateľ je použitý aj na inom mieste, DIA ich berie ako dva rôzne symboly. Nástroj preto porovnáva typové názvy a tým eliminuje problém s `const` a duplikátmi. Používa k tomu 3 kolekcie:

1. `Symbols` – zoznam inštancií triedy `Symbol`; obsahuje len unikátne symboly (napr. len jeden variant z dvojice `int` a `const int`)
2. `SymbolIds` – mapovanie identifikačného čísla symbolu DIA na inštanciu triedy `Symbol` (slúži na rýchly prevod symbolu DIA na inštanciu triedy `Symbol`)
3. `SymbolTypes` – mapovanie typového názvu symbolu na inštanciu triedy Symbol (používa sa na elimináciu duplikátov pri zaraďovaní nových symbolov)

## 4.6 Typový názov

Typy v C a C++ sú reprezentované typovým názvom. Nástroj ich preto taktiež zobrazuje. Pri nájdení a zaradení nového symbolu do zoznamu symbolov dochádza najprv k odvodeniu jeho typového názvu a následne k jeho zjednodušeniu. Tieto kroky sú popísané v nasledujúcich podkapitolách.

### 4.6.1 Odvodenie typového názvu

Rozhranie DIA poskytuje explicitne typové názvy len pri triedach a enumeračných typoch. Pri zvyšných typoch (pri skalárnych typoch, ukazovateľoch, poliach a funkciách) je potrebné typový názov rekurzívne určiť a zároveň dodržať spôsob zápisu typov v jazykoch C a C++. Napríklad pri ukazovateľoch je výsledný typový názov tvorený spojením znaku `*` a názvu ukazovaného typu (ktorým môže byť opäť ukazovateľ). Pre zmenu priority sa používajú jednoduché zátvorky.

Ako bolo spomenuté v kapitole 4.5 nástroj ignoruje konštantnosť typov. To znamená, že symbol DIA, ktorý reprezentuje v C a C++ typ `const int` bude mať v nástroji typový názov `int`.

Typový názov funkcie je nutné explicitne vytvoriť spojením typových názvov jej parametrov. Pri parametroch sa však konštantnosť uplatňuje, preto sú funkcie s parametrami `int*` a `const int*` rôzne.

### 4.6.2 Zjednodušenie typového názvu

Na obrázku 4-10 sa nachádza krátky úsek kódu. Premenná `myvar` je šablónou. Ako argument šablóny je použitý typ z knižnice STL. Rozhranie DIA vráti pre premennú `myvar` jej kompletný typový názov `MyClass<std::_Vector_iterator<int,std::allocator<int>>>`.

```
template <class T>
class MyClass
{
};
MyClass<std::vector<int>::iterator> myvar;
```

**Obrázok 4-10: Šablóna**

Za normálnych okolností by bol tento neprívetivý názov zobrazený aj v rámci (Visual Studio to tak aj robí). V nástroji však bol implementovaný jednoduchý parser, ktorý typový názov rekurzívne rozdelí na jeho zložky a umožní externým vizualizérom tieto zložky zmeniť (zjednodušiť). V tomto prípade by boli postupne zjednodušené podreťazce:

1. `std::allocator<int>` , s výsledkom *N*
2. `std::_Vector_iterator<int,`*N*`>`, s výsledkom *M*
3. `MyClass<M>` , s výsledkom *P*

Výsledný reťazec *P* sa nakoniec uloží do atribútu `TypeString` do príslušnej inštancie triedy `Symbol`. Parser rozpoznáva len identifikátory a znaky `<`, `>`. Všetko ostatné preskakuje. Sleduje zátvorky `(`, `)`, `[`, `]`, `<`, `>` a pamätá si aktuálne zanorenie. Typové názvy pochádzajúce z rozhrania DIA sú syntakticky správne, takže sleduje len stupeň zanorenia.

Samotný parser je implementovaný v triede `TypestringSimplifier`. Vizualizéry dostávajú šancu zjednodušiť zložky typového názvu cez metódu `GetTypeOverride` v rozhraní `IVisualizer`. Môžu k tomu použiť pomocnú triedu `PatternMatcher`, ktorá porovnáva typový názov s daným vzorom. Vzor je tiež reťazcom a v tomto prípade by mohol vyzerať napríklad takto: `std::_Vector_iterator<#,std::allocator<#>>`. Znaky # plnia podobnú úlohu ako regulárny výraz `.+`. Hlavným rozdielom je, že opäť sledujú zanorenie definované zátvorkami. Externý vizualizér potom už len získa reťazce pod znakmi # (v tomto prípade `int`) a vyskladá zjednodušený názov `std::vector<int>::iterator`.

## 4.7 Vstavané vizualizéry

Nástroj obsahuje sadu vstavaných vizualizérov, ktoré pokrývajú štandardné fragmenty jazykov C a C++, napr. skalárne typy, ukazovatele, polia a pod. Tvoria ich hlavne vizualizéry pre druhy symbolov z tabuliek 2-1 a 2-2. Je ich však menej ako polovica oproti všetkým druhom. Zvyšné druhy sa nepoužívajú pri jazykoch C a C++, alebo nepotrebujú mať obraz na ploche. Nasledujúce podkapitoly popisujú spôsoby zobrazenia jednotlivých fragmentov jazykov C a C++.

### 4.7.1 Objekty a premenné

Na obrázku 4-11 sa nachádza krátky program. Premenná `myvar` je štruktúrou, ktorá v seba obsahuje dve vnorené štruktúry. Zobrazenie tejto premennej nástrojom je ukázané na obrázku 4-12. Inicializácia premenných `a`, `b`, `c` hodnotami 3, 4, 5 je kvôli prehľadnosti vynechaná.

Aj keď to z obrázka nie je zrejmé v skutočnosti sa v diagrame nachádzajú skryté rámce, ktoré zastupujú premenné `myvar`, `myvar1`, `myvar2`, `a`, `b`, `c`. Tieto sú ale v tomto prípade zakryté objektmi, ktoré sa v týchto premenných nachádzajú. Toto striedanie premenných a objektov sa deje v celom systéme.

V niektorých prípadoch sú ale rámce zastupujúce premenné predsa len viditeľné. Zmenou typu `MyClass` na `union` (obrázok 4-13) nastane situácia zobrazená na obrázku 4-14. Premenné `a` a `b` obsahujú identický objekt, ktorý nástroj automaticky premiestni do dočasnej skupiny a medzi zúčastnenými rámcami vytvorí vzťah kompozície.

Otázkou je, či by nebolo možné premenné `a` a `b` a ich obsah jednoducho zobrazovať oddelene. Zásadný problém by ale nastal v prípade existencie ukazovateľa na túto premennú. Orientovaná hrana

reprezentujúca ukazovateľ by jednoducho nemala jednoznačný cieľ. Taktiež by museli existovať rámce s totožným `OID`, čo by situáciu skomplikovalo.

Druhou otázkou je prečo vôbec existuje striedanie rámcov pre premenné a objekty v nich. Odpoveď vyplýva z obrázka 4-14. Jeden objekt môže mať viac názvov. Navyše samotné rozhranie DIA je postavené na oddelení premenných a ich typov.

Celkovo platí, že zatiaľ čo premenné poskytujú názov, objekty poskytujú hodnotu. Dôvod, prečo na obrázku 4-12 objekt s hodnotou 3 obsahuje názov je čisto implementačný. Ak rámec nemá vlastný názov, pohľad mu priradí názov predchodcu (v tomto prípade skrytý rámec pre premennú). Podobne na obrázku 4-14 má rámec pre objekt s hodnotou 4 názov priradený z predchodcu (v tomto prípade dočasná skupina, ktorá má ako názov vždy adresu objektu).

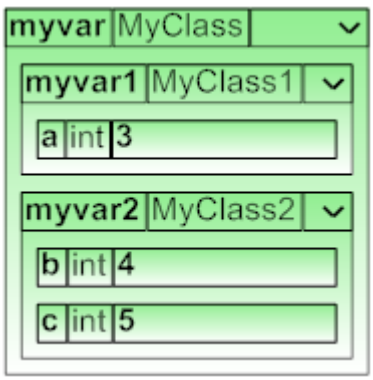
```
struct MyClass1
{
    int a;
};

struct MyClass2
{
    int b;
    int c;
};

struct MyClass
{
    MyClass1 myvar1;
    MyClass2 myvar2;
};

MyClass myvar;
```

**Obrázok 4-11: Štruktúra**




**Obrázok 4-12: Zobrazenie štruktúry**

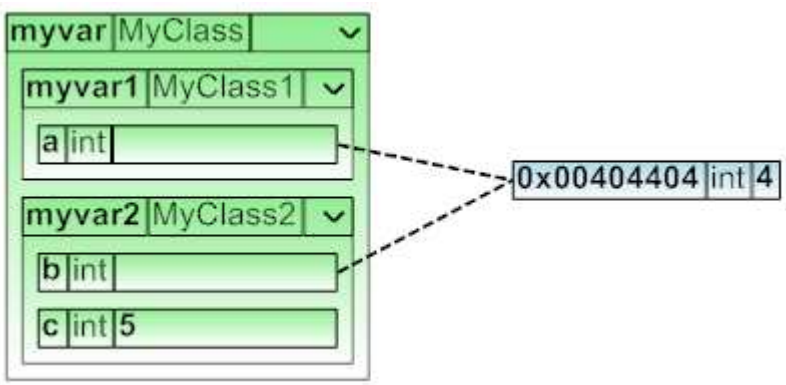
```
struct MyClass1
{
    int a;
};

struct MyClass2
{
    int b;
    int c;
};

union MyClass
{
    MyClass1 myvar1;
    MyClass2 myvar2;
};

MyClass myvar;
```

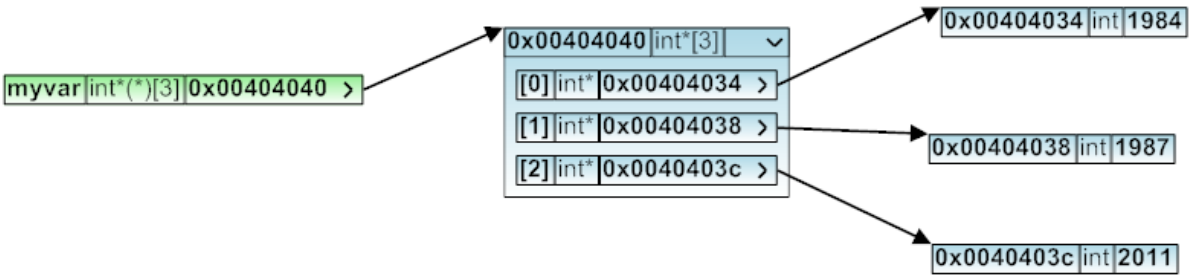
**Obrázok 4-13: Typ union**




**Obrázok 4-14: Zobrazenie typu union**

### 4.7.2 Ukazovatele a polia

Obrázok 4-15 ukazuje akým spôsobom nástroj zobrazuje ukazovatele a polia. Pri poliach opäť platí, že medzi objektom poľa a objektmi prvkov sa nachádzajú skryté rámce pre indexy poľa. Následne rámce pre objekty prvkov poľa zdedia názov (textová podoba indexu) z týchto skrytých rámcov. `OID` skrytých rámcov pre indexy sa rovná `OID` celého poľa. Líšia sa práve indexom v `OID`. Samotné pole má index 0, nultý prvok poľa má index 1, prvý prvok 2, atď. Viacrozmerné polia sú implementované rekurzívne a preto ich nástroj samozrejme podporuje tiež.



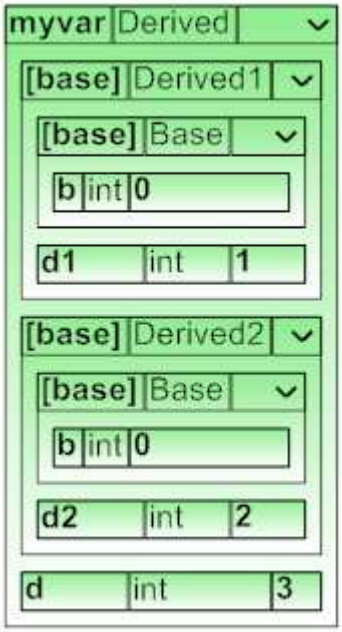

**Obrázok 4-15: Ukazovatele a polia**

### 4.7.3 Dedičnosť

Na obrázku 4-16 sa nachádza jednoduchý príklad dedičnosti. Nástroj zobrazuje túto situáciu tak ako na obrázku 4-17. Na obrázku 4-18 sa nachádza modifikácia príkladu s použitím virtuálnej dedičnosti. Objekt virtuálne dedenej triedy sa v každom objekte nachádza len raz, čo korešponduje so zobrazením na obrázku 4-19. Na obrázku 4-19 je opäť vidno rámec premennej (tentoraz v úlohe bázovej triedy), ktorý oddeľuje dva objekty. Ten bol na obrázku 4-17 skrytý.

```
class Base
{
      int b;
};
class Derived1 : Base
{
      int d1;
};
class Derived2 : Base
{
      int d2;
};
class Derived : Derived1, Derived2
{
      int d;
};
Derived myvar;
```

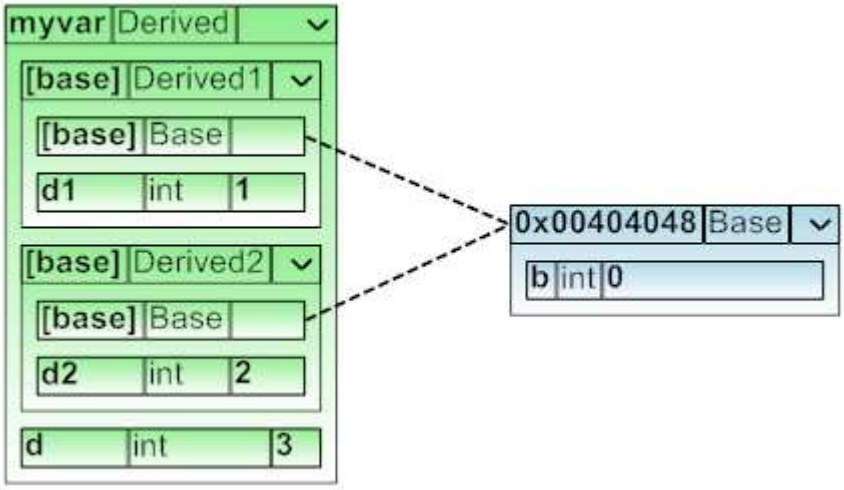
**Obrázok 4-16: Dedičnosť**




**Obrázok 4-17: Zobrazenie dedičnosti**

```
class Base
{
      int b;
};
class Derived1 : virtual Base
{
      int d1;
};
class Derived2 : virtual Base
{
      int d2;
};
class Derived : Derived1, Derived2
{
      int d;
};
Derived myvar;
```

**Obrázok 4-18: Virtuálna dedičnosť**

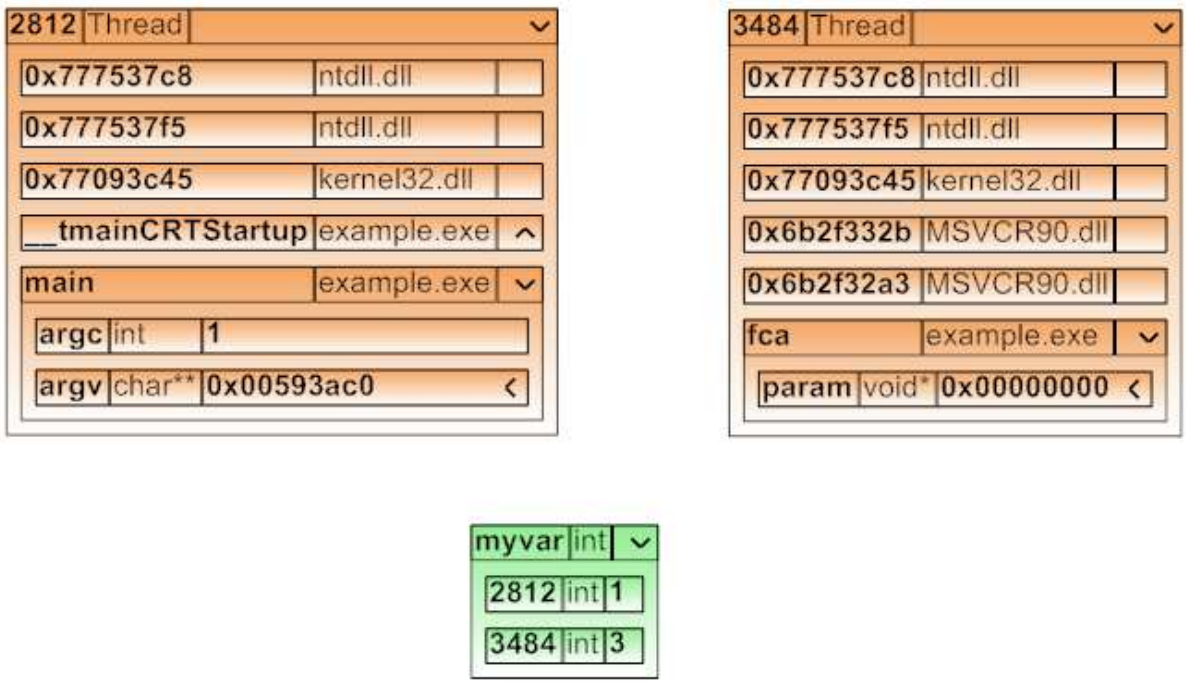



**Obrázok 4-19: Zobrazenie virtuálnej dedičnosti**

### 4.7.4 Vlákna a procedúry

Na obrázku 4-20 je načrtnutý spôsob akým nástroj zobrazuje vlákna a funkcie na ich zásobníkoch. Číslo v ľavom hornom rohu vlákna je jeho identifikačné číslo. Vlákna v nástroji dodržiavajú zaužívanú konvenciu a rastú „smerom dole“.

Na obrázku je ďalej zobrazená premenná TLS (kapitola 2.3.2). Ide o premennú ktorá bola deklarovaná spôsobom na obrázku 4-21. Je to vlastne statická premenná, ktorá má samostatnú inštanciu pre každé vlákno v procese. Názov konkrétnej inštancie je tvorený identifikačným číslom vlákna. Pri premenných TLS sa index v `OID` skrytých rámcov nastavuje na číslo vlákna.




**Obrázok 4-20: Vlákna, funkcie a premenná TLS**

```
__declspec(thread) int myvar;
```

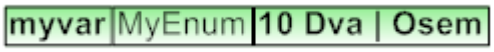
**Obrázok 4-21: Spôsob deklarácie premennej TLS**

### 4.7.5 Ďalšie vizualizéry

Zvyšným vizualizérom už nebude venovaná samostatná podkapitola. Prvým je enumeračný typ `enum` (obrázok 4-22). Nástroj zobrazí jeho hodnotu aj ako kombináciu konštánt v ňom. Pri nejednoznačnosti nezobrazuje duplicitné hodnoty.

Keďže ukazovateľ môže ukazovať aj na funkciu, je potrebné ich zobrazovať tiež (obrázok 4-23). Pri funkciách hrá úlohu skrytej premennej konkrétna funkcia v kóde. Jej následníkom je typ funkcie na danej adrese.

Bitové polia sa zobrazujú podobne ako štruktúry.




**Obrázok 4-22: Enumeračný typ**


**Obrázok 4-23: Funkcia**

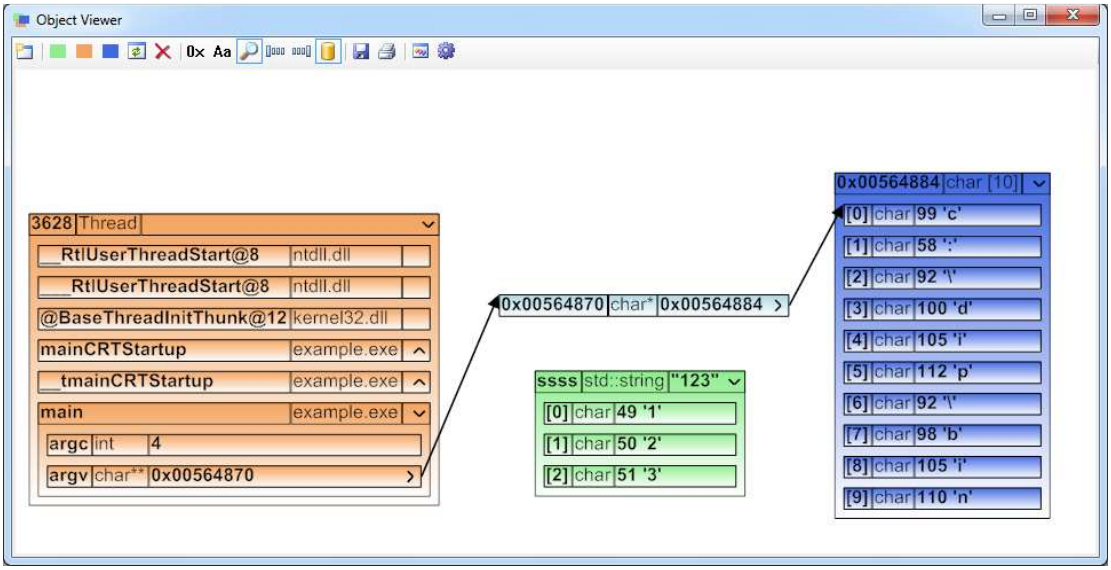
Nástroj si v niektorých prípadoch poradí aj so skalárnymi premennými a parametrami, ktoré kompilátor kvôli optimalizácii umiestni len do niektorého registra. Túto situáciu je možné nasimulovať napríklad označením funkcie ako `__fastcall` a zapnutím optimalizácií. V takomto prípade sa ale hodnota premennej zobrazí v rámci pre premennú a nie v rámci pre objekt. Je to tak preto, pretože premenná v registri nemá žiadnu adresu a adresa v `OID` musí byť platná.

## 4.8 Jadro

Na obrázku 4-24 sa nachádza ukážka samostatne spusteného jadra, resp. jeho viditeľnej časti (Viewer). Na ploche pohľadu je vidieť všetky štyri druhy skupín. Zelenú farbu majú skupiny reprezentujúce globálne / statické premenné programu. Hnedou farbou sú zobrazené zásobníky vlákien. Tmavomodrá farba reprezentuje dynamické skupiny. Typ a adresu dynamickej skupiny určuje užívateľ. V prípade, že sa k nejakému objektu nedá dostať cez predchádzajúce skupiny je dynamická skupina

poslednou možnosťou ako ho zobraziť. Dočasné skupiny majú svetlomodrú farbu. Sú viditeľné len do tej doby, pokým na ne existuje odkaz z inej viditeľnej skupiny.

V pravom hornom rohu každého rámca sú tlačidlá na rozbalenie a nalinkovanie rámca. Skupiny ako celky je možné ľubovoľne rozmiestniť na ploche. Koliesko myši slúži na zmenu mierky pohľadu.




**Obrázok 4-24: Jadro**

Nástroj obsahuje v hornej časti panel nástrojov. Obsahuje tlačidlá, pomocou ktorých je možné vykonať nasledujúce úkony:

1. zobraziť okno s novým pohľadom
2. vložiť na plochu globálnu premennú, vlákno, alebo dynamickú skupinu
3. obnoviť model a pohľad (napr. po zmene kontextu procesu)
4. odstrániť z plochy všetky rámce
5. prepínať medzi zobrazením čísel v decimálnej a hexadecimálnej sústave a medzi kapitálkami a malými písmenami
6. zapnúť / vypnúť externé vizualizéry
7. zapnúť / vypnúť zobrazovanie zarážok rend a end pri kontajneroch
8. povoliť alebo zakázať skrytú pamäť rámcov (ak je táto funkcia zapnutá, rámce si pamätajú svoj stav aj po odstránení z plochy; pri opätovnom pridaní sa ich stav obnoví)
9. uložiť obraz pohľadu do bitmapového súboru a tlač
10. zobraziť zoznam modulov
11. zobraziť dialóg s nastaveniami

Nové skupiny sa na plochu vkladajú pomocou dialógov. Dialógy pre globálne skupiny, skupiny pre vlákna a dynamické skupiny sa nachádzajú na obrázkoch 4-25, 4-26 a 4-27. Je možné vybrať viac

skupín naraz. Textové pole v hornej časti dialógov sa používa na spresnenie vyhľadávania. Pri dynamickej skupine užívateľ musí zadať aj adresu a voliteľne veľkosť poľa.

Kliknutie pravého tlačidla myši na rámec zobrazí ďalšie možnosti. Z tade je možné zobraziť a zmeniť hodnotu rámca v samostatnom dialógu (obrázok 4-28). Ukazovatele je možné meniť jednoduchým presunom orientovanej hrany na iný rámec, poprípade na plochu (vtedy sa hodnota ukazovateľa nastaví na 0).

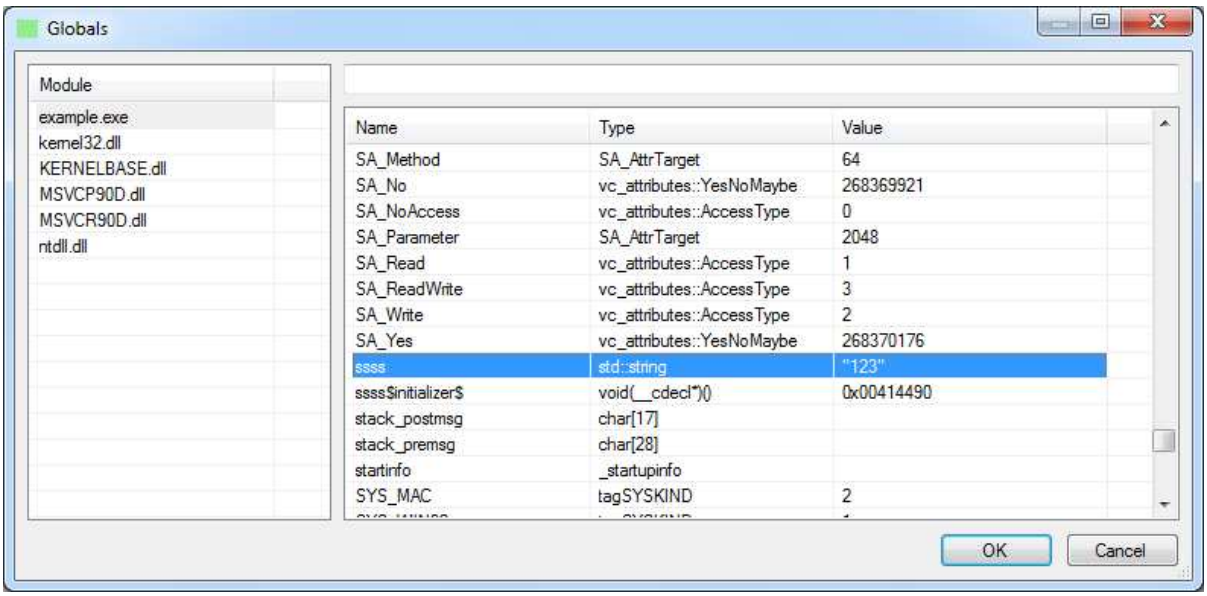
Rámce obsahujúce objekty sa dajú prichytiť na plochu (angl. *pin*). Pri tom sa pre ne vytvorí dynamická skupina. To je vhodné najmä pri dočasných skupinách, kedy užívateľ nechce riskovať schovanie dočasnej skupiny kvôli strate dostupnosti.

Obrázok 4-29 obsahuje dialóg so zoznam modulov. Pri každom z nich je cesta k súboru so symbolickými informáciami ak bol vyhovujúci súbor nájdený. Pravým tlačidlom myši je možné manuálne tieto súbory otvoriť alebo zatvoriť. Nástroj pri vyhľadávaní súborov prehľadáva niekoľko adresárov. Primárne sa berie do úvahy adresár, v ktorom sa nachádza daný modul. V dialógu nastavení je možné nastaviť ďalšie cesty. Tie môžu ukazovať aj na server so symbolmi (symbol server). Knižnica msdia90.dll na komunikáciu s ním používa knižnicu symsrv.dll, ktorá je taktiež súčasťou nástroja.

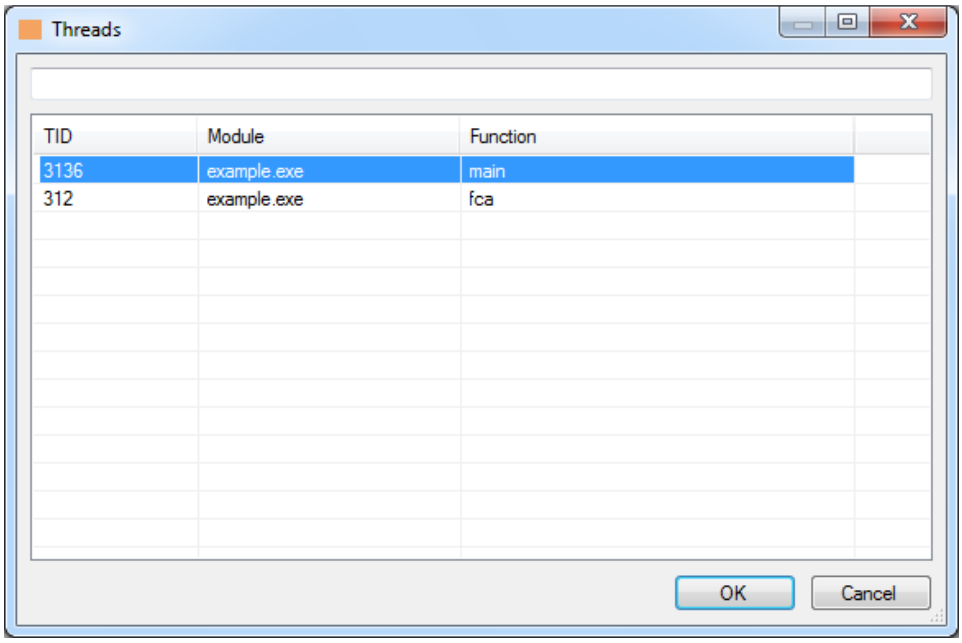
Jadro sa spúšťa z príkazového riadku jedným z nasledujúcich spôsobov:

```
Kernel -pid <PID>
Kernel -name <NAME>
```

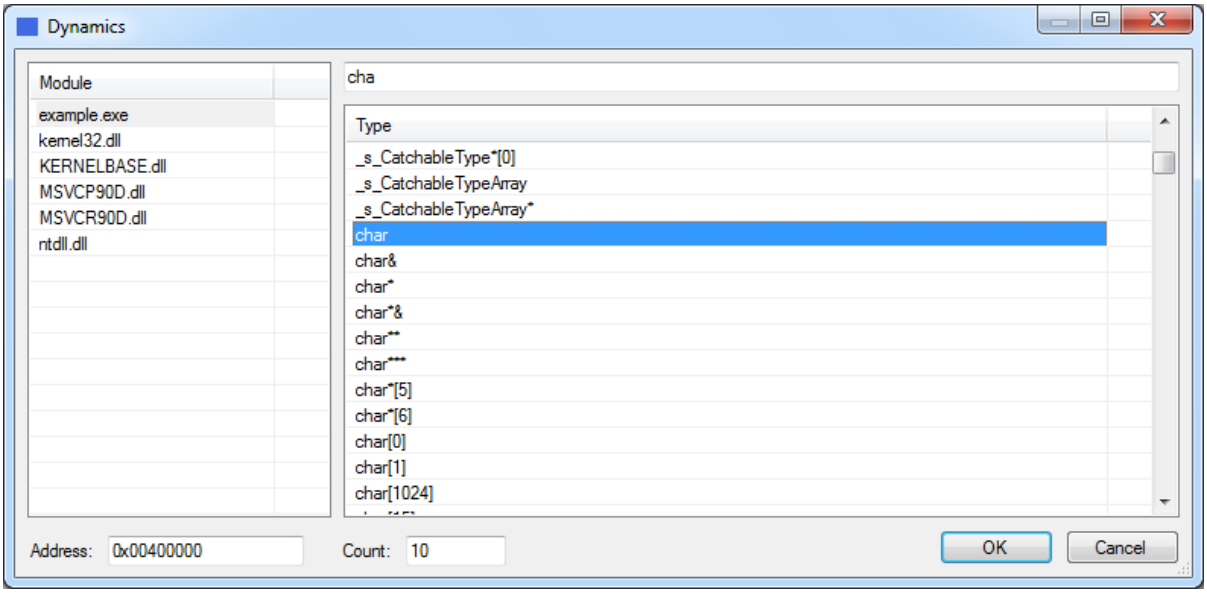
V prvom prípade sa použije proces s daným identifikačným číslom, v druhom prípade sa proces nájde podľa názvu (v prípade programu example.exe je to example).



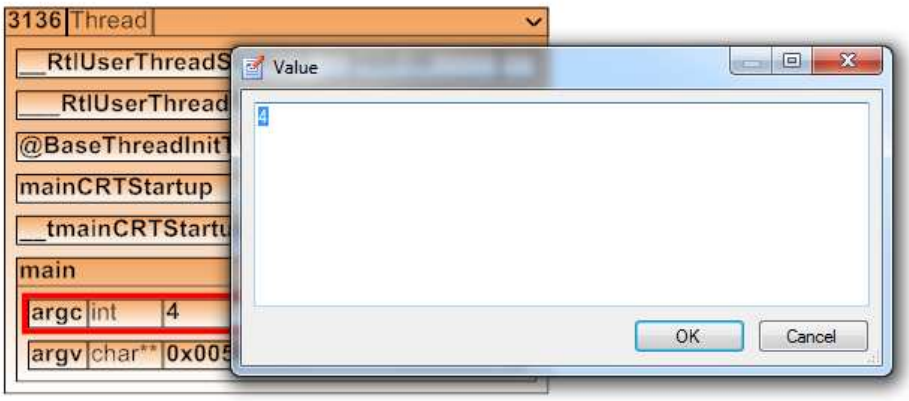

**Obrázok 4-25: Vloženie globálnej skupiny**



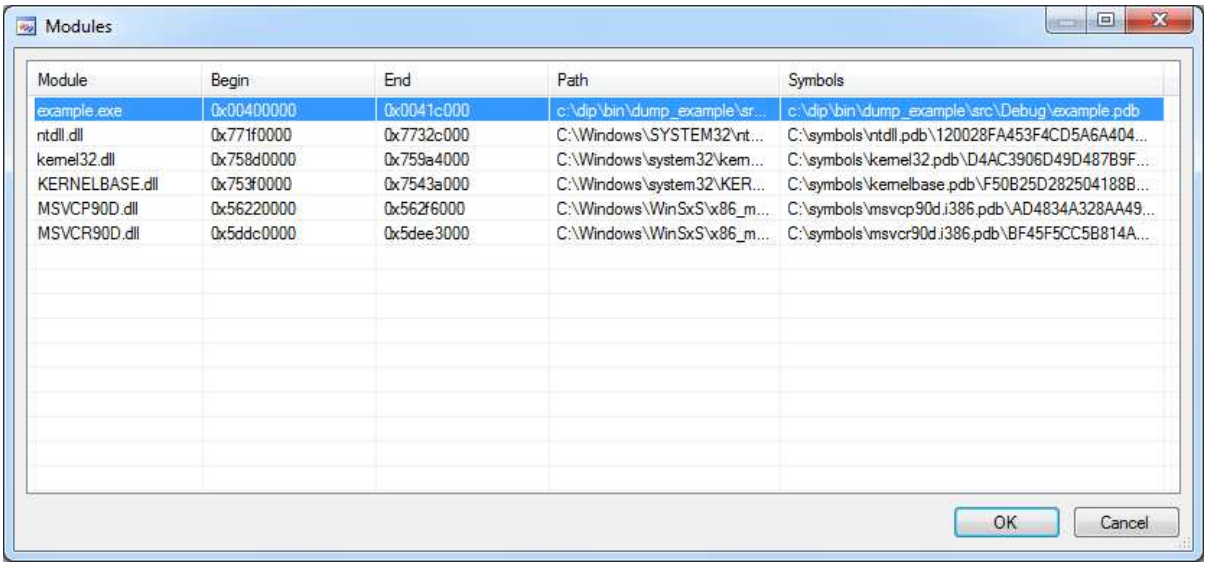

**Obrázok 4-26: Vloženie skupiny pre vlákno**




**Obrázok 4-27: Vloženie dynamickej skupiny**




**Obrázok 4-28: Zmena hodnoty premennej**




**Obrázok 4-29: Zoznam modulov**

## 4.9 Integrácia do prostredia Visual Studio

Vývoj rozšírení existujúcich aplikácií nie je jednoduchou záležitosťou a preto sa ukázalo, že umiestnenie prakticky celej logiky nástroja do samostatnej komponenty bolo správne. Pri vývoji bol použitý sprievodca vytvárania rozšírení a do výsledku bol len pridaný kód, ktorý načíta jadro.

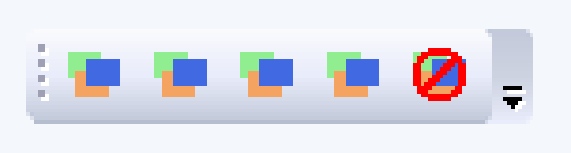
Rozšírenie po prvom spustení vytvorí samostatný panel nástrojov (obrázok 4-30). Z tade je možné zobraziť jeden zo štyroch pohľadov. Rovnaký systém pevného počtu okien používa Visual Studio aj pri vstavaných oknách, napr. pri nástroji Watch.

Po spustení prostredia je nástroj implicitne deaktivovaný. Zobrazením pohľadu sa automaticky aktivuje. Deaktivuje sa stisnutím posledného tlačidla v paneli nástrojov alebo cez manažér rozšírení (Add-in Manager). Tam je možné ho nastaviť aj tak, aby sa aktivoval vždy po štarte.

V aktívnom stave nástroj sleduje ladenie a pri pozastavení programu obnovuje plochy všetkých pohľadov. Ak ho užívateľ nepoužíva môže ho deaktivovať a tým zvýšiť rýchlosť zvyšných komponentov prostredia.

Samotné okná pohľadov vyzerajú podobne ako pri samostatnom spustení nástroja. Sú však súčasťou prostredia a preto sa s nimi pracuje rovnako ako so vstavanými oknami. Zatvorenie okien spôsobí len ich skrytie a ich obsah zostane nezmenený.

Rozšírenie je implementované v jazyku C# pre prostredie Visual Studio 2008.



**Obrázok 4-30: Panel nástrojov**

## 4.10 Externé vizualizéry

Nástroj očakáva externé vizualizéry vo forme knižníc, ktoré sa nachádzajú v podadresári plugins. Knižnice implementujú rozhranie `IPlugin` definované v jadre. Jadro volá pri inicializácií metódu `Load` tohto rozhrania a knižnice na to reagujú registrovaním svojich podporovaných vizualizérov v jadre.

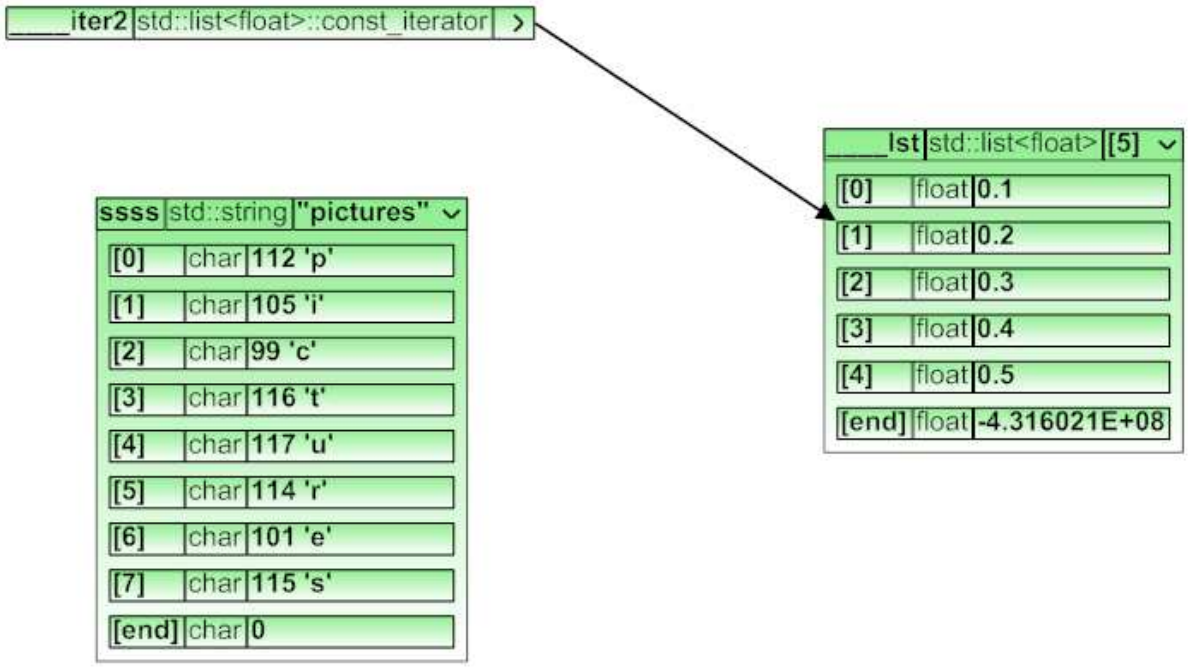
Súčasťou implementácie je aj ukážková implementácia externých vizualizérov. Tie boli vytvorené pre často používané typy z knižnice STL. Jedná sa o kontajnery `list`, `vector`, `string` (resp. `basic_string`), `set`, `multiset`, `map`, `multimap` a o ich konštantné a reverzné iterátory.

Externé vizualizéry používajú vstavané na získanie pôvodnej štruktúry objektov. Najčastejšie využívajú metódy `GetValue` a `GetLink` rozhrania `IVisualizer`, ktoré vracajú hodnotu resp. odkazovanú hodnotu objektu resp. ukazovateľa. Následne definujú pozmenené alebo úplne nové chovanie typov.

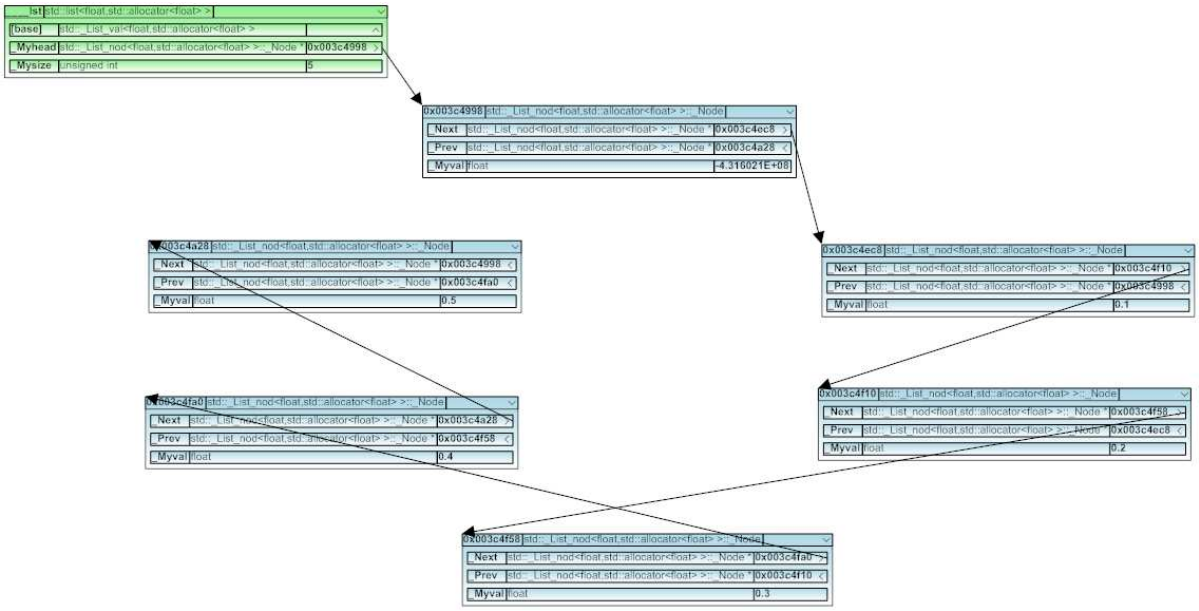
Na obrázku 4-31 je ukážka dvoch kontajnerov a jedného iterátora. V tomto prípade je zároveň povolené zobrazovanie zarážok (prvok [end]), čo je použiteľné napríklad pri vizualizácií algoritmov.

Obrázok 4-32 ukazuje jeden z týchto kontajnerov (`list`) zobrazený s vypnutými vizualizérmi. Obrázok naznačuje, že list je v použitej implementácií STL implementovaný pomocou obojsmerne viazaného zoznamu. V prípade kontajnerov `set` a `map` by napríklad išlo o strom.

Rozšírenie bolo implementované v jazyku C#. Výsledný súbor má názov stl.dll.




**Obrázok 4-31: Kontajnery so zapnutými vizualizérmi**



**Obrázok 4-32: Skutočná štruktúra kontajnera list**

# 4.11 Exploratívne funkcie

V nástroji boli implementované 3 funkcie s exploratívnym charakterom. Ich funkčnosť nie je síce na takej úrovni ako v dynamických jazykoch, ale v jazykoch C a C++ to ani nie je možné.

### 4.11.1 Xrefs

Xrefs (skratka z angl. *cross references*) v tomto kontexte znamená nájdenie všetkých rámcov, ktoré sa nachádzajú v definovanom vzťahu k vybranému rámcu. Prehľadávanie začína na viditeľných a/alebo neviditeľných skupinách. Pri tom sa rozbaľujú vnorené rámce a voliteľne sa sledujú aj ukazovatele. Prehľadávanie sa deje v súlade s algoritmom *Breadth-first search*. V momente keď algoritmus narazí na pôvodný rámec, zobrazí, rozbalí a nalinkuje všetky rámce, ktoré sa nachádzajú na ceste medzi skupinou a rámcom.

Na obrázku 4-33 sa nachádza dialóg s nastavením algoritmu. Vhodným povolením jednotlivých možností sa dosahujú obidva prípady použitia na nájdenie skupiny rámca a referencií naň.


**Obrázok 4-33: Dialóg Xrefs**

### 4.11.2 Skok na definíciu

Nástroj pri niektorých rámcoch umožňuje otvoriť miesto v zdrojom kóde, ktoré obsahuje definíciu typu alebo premennej rámca. Používa sa na to metóda `CodeElementFromFullName` rozhrania `VCCodeModel`. Je možné do nej vložiť plný názov premennej alebo typu a v niektorých prípadoch si poradí aj so šablónami. Časť funkčnosti súvisiaca s rozhraním `VCCodeModel` sa nachádza v rozšírení prostredia, zvyšok v jadre. Ak sa definíciu nepodarí nájsť (napr. kvôli makrám), nástroj sa pokúsi nájsť definíciu materských rámcov.

### 4.11.3 Vykonanie metódy nad objektom

Nástroj neumožňuje priamo vykonať metódu nad daným objektom. Dokáže však vytvoriť reprezentáciu objektu, ktorú môže užívateľ pomocou schránky operačného systému vložiť do okna Immediate Window a po vpísaní názvu príslušnej metódy, alebo operátora ju vykonať. Príklad takej reprezentácie je napr.: `(*(std::list<float,std::allocator<float> > *)0x004221fc)`. Nie je to vlastne nič iné ako kombinácia typu a adresy.

## 4.12 64 bitové rozšírenie

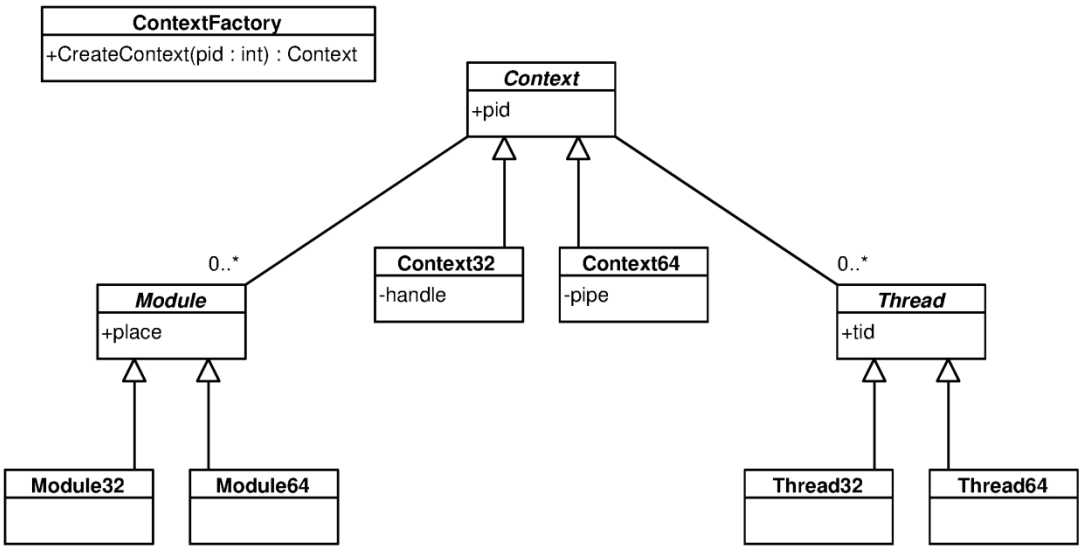
Zapuzdrenie adries pochádzajúcich z ladeného procesu do triedy `Address` sa ukázalo byť správnym rozhodnutím pri rozšírení podpory nástroja pre ladenie 64 bitových aplikácií. Bez toho by hrozilo nebezpečenstvo, že by niekde v implementácii existovalo miesto, kde by sa adresa uchovávala v bežne používanom 32 bitovom type `int`.

Prostredie Visual Studio podporuje ladenie 64 bitových aplikácií. Samotné prostredie však existuje len v 32 bitovej forme. Operačný systém Windows nedovolí ladenie 64 bitovej aplikácie 32 bitovou aplikáciou. Visual Studio tento problém rieši pridaním 64 bitového programu medzi seba a 64 bitovú aplikáciu. S týmto programom komunikuje pomocou niektorej metódy na komunikáciu medzi procesmi. V prípade lokálneho ladenia používa pomenovanú rúru (*named pipe*).

Rovnaký princíp prístupu k 64 bitovým aplikáciám bol použitý aj v nástroji. Bola vytvorená 64 bitová spustiteľná aplikácia srv64, ktorá slúži ako prostredník medzi ladeným programom a nástrojom. Nástroj s ňou komunikuje pomocou pomenovanej rúry.

Zároveň bola prevedená refaktorizácia balíka Context. Výsledný diagram tried sa nachádza na obrázku 4-34. Spoločná funkcionalita zostala v pôvodných triedach. Zatiaľ čo triedy pre 32 bitové aplikácie priamo otvárajú ladený proces, triedy pre 64 bitové aplikácie sa pripájajú na pomenovanú rúru a cez ňu posielajú všetky požiadavky procesu srv64. Trieda `Context64` zároveň spúšťa a ukončuje beh aplikácie srv64.

Aplikácia srv64 bola pôvodne implementovaná v jazyku C#. Z neznámej príčiny sa však nechovala rovnako v konfigurácii Debug a Release a preto bola prepísaná do jazyka C++. Konkrétne išlo o funkciu `GetThreadContext` operačného systému, ktorá v konfigurácii Release neprebehla správne.




**Obrázok 4-34: Refaktorizácia balíka Context**

## 4.13 Inštalácia

Poslednou časťou implementácie bolo vytvorenie inštalačného balíka, keďže je nástroj tvorený niekoľkými súbormi. Súbory v balíku sú zhrnuté v tabuľke 4-1. Inštalačný balík bol vytvorený v prostredí Visual Studio a má názov ObjectViewer.msi.

| **Súbor** | **Popis** |
|---|---|
| Kernel.exe | jadro aplikácie |
| msdia90.dll | rozhranie DIA |
| symsrv.dll | prístup k serveru so symbolmi |
| srv64.exe | ladenie 64 bitových aplikácií |
| stl.dll | vizualizéry pre typy z STL |
| UMD.HCIL.Piccolo.DLL | knižnica Piccolo2D |
| ObjectViewerAddin.dll | rozšírenie prostredia Visual Studio |
| ObjectViewerAddin.resources.dll | ikony používané v rozšírení |
| ObjectViewer2005.AddIn | registračný súbor pre Visual Studio 2005 |
| ObjectViewer2008.AddIn | registračný súbor pre Visual Studio 2008 |
| ObjectViewer2010.AddIn | registračný súbor pre Visual Studio 2010 |
| unregister.vbs | odstráni tlačidlá vytvorené rozšírením z prostredia |

**Tabuľka 4-1: Zoznam súborov**

Nástroj bol testovaný na niekoľkých operačných systémoch a prostrediach. Medzi minimálne systémové požiadavky patrí operačný systém Microsoft Windows XP (32 alebo 64 bitová verzia), vývojové prostredie Microsoft Visual Studio 2005 a .NET Framework 3.5.

# 5 Zhodnotenie dosiahnutých výsledkov

Nástroj počas vývoja pravidelne podliehal testovaniu. Pri implementovaní vizualizérov pre knižnicu STL bol použitý na skúmanie a pochopenie štruktúry a chovania jej kontajnerov a iterátorov. Následne bol otestovaný na niekoľkých náhodne vybraných jednoduchých programoch. Potom bola preskúmaná časová zložitosť programu v rôznych situáciách.

Dialóg dynamických skupín zobrazuje všetky typy. Čas potrebný na ich načítanie je závislý od ich počtu. Graf na obrázku 5-1 zobrazuje závislosť tohto času od počtu typov. Krivka 1 reprezentuje prvotné zobrazenie zoznamu typov a krivka 2 každé ďalšie zobrazenie zoznamu. Z grafu je zrejmé, že čas potrebný na zobrazenie typov v dialógu dynamických skupín je prvýkrát väčší. To vyplýva z faktu, že pri nájdení nového symbolu jeho identifikačné číslo neexistuje v modeli a je potrebné ho nájsť na základe jeho typového názvu.

Ďalej bola preskúmaná závislosť času obnovenia plochy od veľkosti testovaného poľa (a teda od počtu viditeľných rámcov). Obrázok 5-2 obsahuje graf tejto závislosti. Obsahuje 2 krivky podľa toho či ide o bezprostredné obnovenie po zobrazení poľa (1), alebo už medzitým prebehlo obnovenie (2). Z grafu je vidno, že aj pri neúmerne veľkom počte zobrazených prvkov je čas stále únosný.

Vytvorený nástroj je dobrým základom pre ďalší vývoj. Aj keď podporuje 32 aj 64 bitové aplikácie, bolo by vhodné pridať podporu aj pre iné používané platformy (napr. .NET a Java). Pri týchto platformách nie je potrebné riešiť problémy charakteristické pre jazyky C a C++, napr. neplatné ukazovatele. Zatiaľ čo pamäťový model jazykov C a C++ je priamočiary, pri platformách s virtuálnym strojom by bolo pravdepodobne potrebné komunikovať s ním. To znamená, že funkcionalita nástroja by závisela od jeho možností. S pridaním podpory pre ďalšie platformy by bolo potrebné upraviť návrh striktným oddelením platformovo nezávislej časti od závislej.

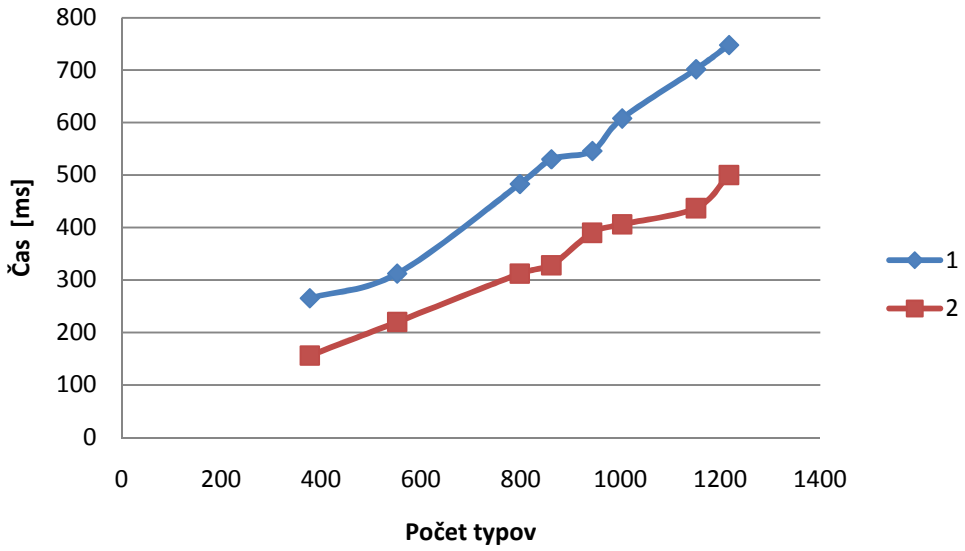
Vizualizéry pre typy z STL fungujú dobre pokiaľ majú viditeľné objekty týchto typov platné hodnoty, poprípade nemajú žiadne hodnoty (objekt sa nachádza v uvoľnenej pamäti). V prípade, že obsah objektov tvoria náhodné hodnoty, to môže viesť k zaseknutiu, alebo k pádu nástroja. Tento problém bude potrebné vyriešiť. Veľmi jednoduchou a priamočiarou možnosťou by mohlo byť obmedzenie počtu prvkov kontajnerov na nejaký pevný počet, ktorý by v prípade potreby bolo možné zmeniť v dialógu nastavení. Ani to však nemusí stačiť. V programe môže existovať kontajner, ktorý v sebe obsahuje vnorený kontajner. V takom prípade by bolo možné napríklad obmedziť celkový počet rámcov. Po prekročení maximálneho počtu by mal byť užívateľ na túto situáciu upozornený.

V neposlednom rade by bolo vhodné do nástroja pridať ďalšie exploratívne funkcie. Keď už dokáže skočiť na definíciu typu alebo premennej, mohol by zobrazovať komentár typu alebo premennej v podobe bublinovej nápovedi nad rámcom.

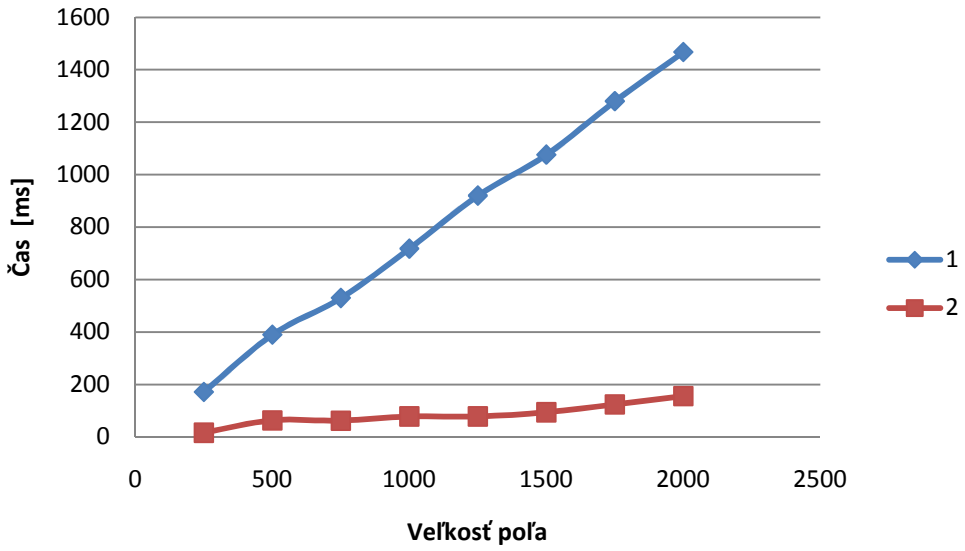
Vytvorený nástroj si kladie za cieľ byť exploratívnym doplnkom vstavaných nástrojov prostredia Visual Studio, ktoré zobrazujú obsah premenných v jednoduchej tabuľke. Vo väčšine prípadov

chce užívateľ rýchlo zistiť hodnotu práve používaných premenných a na túto činnosť sa jednoducho hodia viac. Vstavané nástroje tiež vyberajú do tabuliek len niektoré premenné podľa kontextu. Na druhej strane Object Viewer je možné použiť pri práci so širším kontextom. Grafické zobrazenie ukazovateľov sa veľmi hodilo pri skúmaní štruktúry typov z STL. Typy kontajnerov a iterátorov nástroj zobrazuje v zjednodušenej podobe.

Prax ukázala, že na bežné používanie je vhodné mať monitor s väčším rozlíšením, poprípade viac monitorov a nástroj mať zobrazený na samostatnom monitore.




**Obrázok 5-1: Graf závislosti časovej náročnosti od počtu typov**




**Obrázok 5-2: Graf závislosti časovej náročnosti od veľkosti poľa**

# 6 Záver

Táto diplomová práca sa venovala nástrojom, ktoré sa bežne používajú pri exploratívnej editácií zdrojových textov v dynamických jazykoch Smalltalk a Self.

Najprv boli uvedené príklady takých nástrojov. Úlohou bolo si vybrať jeden alebo viac z nich a navrhnúť ich obdobu pre iný jazyk. Mnohé z nich sa však už v bežne používaných vývojových prostrediach nachádzajú. Bola síce možnosť rozšíriť funkčnosť existujúcich, ale to by pravdepodobne neprinieslo originálny, zaujímavý výsledok. Nakoniec bol vybraný nový nástroj Object Viewer. Jeho inšpiráciou sa stalo samotné prostredie jazyka Self. Ako cieľové programovacie jazyky boli zvolené C a C++. Nástroj síce neumožňuje priamo editovať zdrojový kód, ale je možné ho využiť ako podporný nástroj pre túto činnosť. Nasledovala analýza niekoľkých rozhraní, na ktorých bol postavený.

Intelektuálne najťažšou časťou celej práce bol návrh. Pri tom bolo potrebné zohľadniť všetky vlastnosti a špeciality vybraných jazykov. Chyba alebo nedomyslený problém v tejto fáze by viedli ku vzniku situácií, pre ktoré by v nástroji jednoducho neexistovala podpora. Preto bol návrhu venovaný dostatok času.

Vďaka tomu prebiehala implementácia prakticky bez vážnejších problémov a len s minimálnymi zmenami oproti návrhu. Podarilo sa implementovať všetky navrhnuté vlastnosti a funkcie nástroja.

Následne bol nástroj testovaný a boli diskutované dosiahnuté výsledky. V niekoľkých bodoch bol načrtnutý ďalší možný vývoj.

Výsledná aplikácia je pravdepodobne jedinou aplikáciou svojho druhu pre jazyky C a C++. Väčšina podobných nástrojov sa striktne drží statického sveta diagramu tried. Okrem vývoja aplikácií je nástroj použiteľný pri študovaní existujúcich programov a komplikovaných dátových štruktúr a v neposlednom rade aj na výučbu algoritmov.

# Literatúra


[1] ORLEANS, Doug. *Doug's Home Page* [online]. 1998-10-16 [cit. 2011-05-18]. Exploratory Programming with Collaborative Programming Languages. Dostupné z WWW: <http://steak.place.org/dougo/thesis/plan>.

[2] KŘIVÁNEK, Pavel; KŘIVKA, Zbyněk. *Squeak Smalltalk* [online]. 2003-04-11 [cit. 2011-05-18]. Smalltalk. Dostupné z WWW: <http://www.squeak.cz/Squeak/15>.

[3] INGALLS, Daniel. *Computer Science at the UNIVERSITY of VIRGINIA* [online]. 2001 [cit. 2011-05-18]. Design Principles Behind Smalltalk. Dostupné z WWW: <http://www.cs.virginia.edu/~evans/cs655/readings/smalltalk.html>.

[4] KADANSKY, Miriam. *Oracle Labs* [online]. 2006 [cit. 2011-05-18]. Self. Dostupné z WWW: <http://labs.oracle.com/self>.

[5] UNGAR, DAVID; SMITH, RANDALL. SELF: The Power of Simplicity. *OOPSLA '87 Conference Proceedings*. 1987, 22, s. 227-241. Dostupný také z WWW: <http://labs.oracle.com/self/papers/selfPower.ps.gz>.

[6] CHANG, BAY-WEI; UNGAR, DAVID. Experiencing SELF Objects : An Object-Based Artificial Reality. *The Self Papers*. 1990. Dostupný také z WWW: <http://labs.oracle.com/self/papers/ui.ps.gz>.

[7] CICHON, Gordon. *Self Support* [online]. 1997 [cit. 2011-05-18]. What is Self anyways?. Dostupné z WWW: <http://www.self-support.com/index.html>.

[8] BRAGDON, Andrew. *Andrew Bragdon* [online]. 2010 [cit. 2011-05-18]. Code Bubbles. Dostupné z WWW: <http://www.andrewbragdon.com/codebubbles_site.asp>.

[9] KRUGLINSKI, David; SHEPHERD, George; WINGO, Scot. *Programming Microsoft Visual C++*. 5. [s.l.] : Microsoft Press, 1998. 1150 s.

[10] STROUSTRUP, Bjarne. *The C++ Programming Language*. 3. Massachusetts : Addison-Wesley, 1997. 912 s.

[11] *Msdn* [online]. 2011 [cit. 2011-05-18]. CodeModel Interface. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/envdte.codemodel(v=vs.80).aspx>.

[12] *Msdn* [online]. 2011 [cit. 2011-05-18]. Debug Interface Access SDK. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/x93ctkx8(v=VS.80).aspx>.

[13] AHO, Alfred; LAM, Monica; SETHI, Ravi; ULLMAN Jeffrey. *Compilers : Principles, Techniques, and Tools*. 2. Boston : Addison Wesley, 2006. 1009 s.

[14] *Msdn* [online]. 2011 [cit. 2011-05-18]. Visualizers. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/zayyhzts.aspx>.

[15] *Msdn* [online]. 2011 [cit. 2011-05-18]. Expression Evaluator. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb162318.aspx>.

[16] *Piccolo2D* [online]. 2008 [cit. 2011-05-18]. A Structured 2D Graphics Framework. Dostupné z WWW: <http://www.piccolo2d.org/index.html>.

[17] *Msdn* [online]. 2011 [cit. 2011-05-18]. Create and Analyze a Mini-Dump File in Windows. Dostupné z WWW: <http://msdn.microsoft.com/en-us/visualc/bb738099.aspx>.

# Zoznam príloh

A. CD nosič s elektronickou podobou diplomovej práce, zdrojovými kódmi, programovou dokumentáciou, binárnymi súbormi a demonštračnými videami.